*”Of chess it has been said that life is not long enough for it, but that is the fault of life, not chess.” - Irving Chernev*

# Implementation of Chess bot through Reinforcement Learning

**Sohaib Ahmed - 373937**
**Muhammad Usman Bashir - 366171**

## Abstract

This report details the implementation of a chess bot using reinforcement learning (RL). Traditional chess engines like IBM's Deep Blue used brute force computation and heuristic evaluation, which have limitations. RL offers a dynamic approach where an agent learns by performing actions and receiving feedback, aiming to maximize rewards over time.

Key RL concepts in chess include state, action, reward, policy, and value function. Techniques like Alpha-Beta Pruning, Temporal Difference (TD) Learning, and Monte Carlo Tree Search (MCTS) optimize decision-making.

AlphaZero, developed by DeepMind, uses MCTS and deep neural networks to learn from self-play, outperforming traditional engines like Stockfish. Despite computational challenges, the report highlights the potential of RL in creating advanced, self-improving chess engines.

## Introduction

Chess is one of the oldest and most celebrated strategy games in history, offering a profound level of complexity and requiring deep strategic thinking. This game, which involves two players moving pieces on an 8x8 board, has been a subject of fascination not only for human players but also for artificial intelligence (AI) researchers. The objective is to checkmate the opponent's king, a task that demands careful planning and anticipation of the opponent's moves.

### The Evolution of Chess Engines

Chess engines have evolved significantly over the past decades. Early chess programs relied on brute force computation and heuristic evaluation functions to assess board positions. These engines, like IBM's Deep Blue, could search millions of positions per second, relying heavily on human-crafted rules and vast databases of chess openings and endgames.

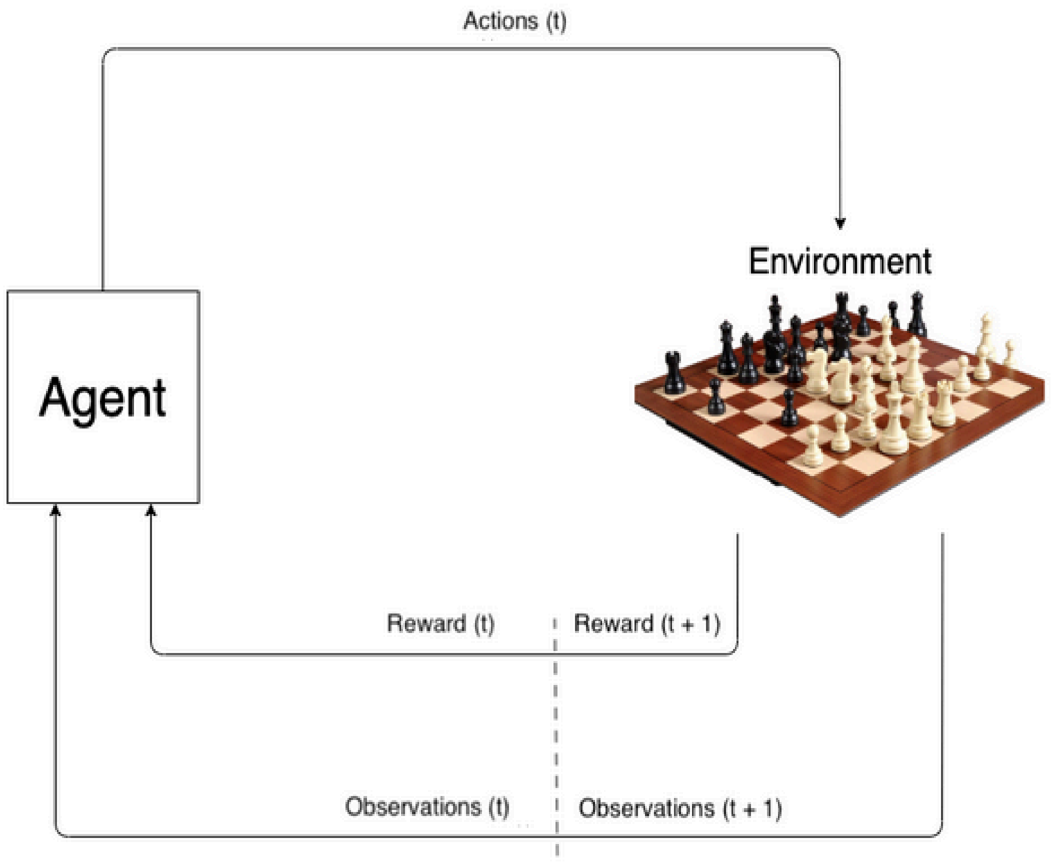
However, these traditional methods have their limitations, primarily because they depend on static knowledge bases and lack the ability to adapt and learn from new situations dynamically. This is where reinforcement learning (RL) comes into play.

## Reinforcement Learning: A Dynamic Approach

Reinforcement learning is a type of machine learning where an agent learns to make decisions by performing actions and receiving feedback from the environment in the form of rewards or penalties. The goal is to maximize cumulative rewards over time. This method is particularly well-suited for games like chess, where the outcome of a move depends not only on immediate consequences but also on future possibilities.

### Key Concepts of Reinforcement Learning in Chess

1. **State:** The configuration of the chessboard at any given time.
2. **Action:** Any legal move a player can make from a given state.
3. **Reward:** The feedback received after making a move. In chess, rewards can be based on capturing pieces, checkmating the opponent, or other positional advantages.
4. **Policy:** The strategy used by the agent to decide the next action based on the current state.
5. **Value Function:** Estimates the expected cumulative reward from a given state, guiding the agent to make decisions that maximize long-term gains.



## Interactive Elements

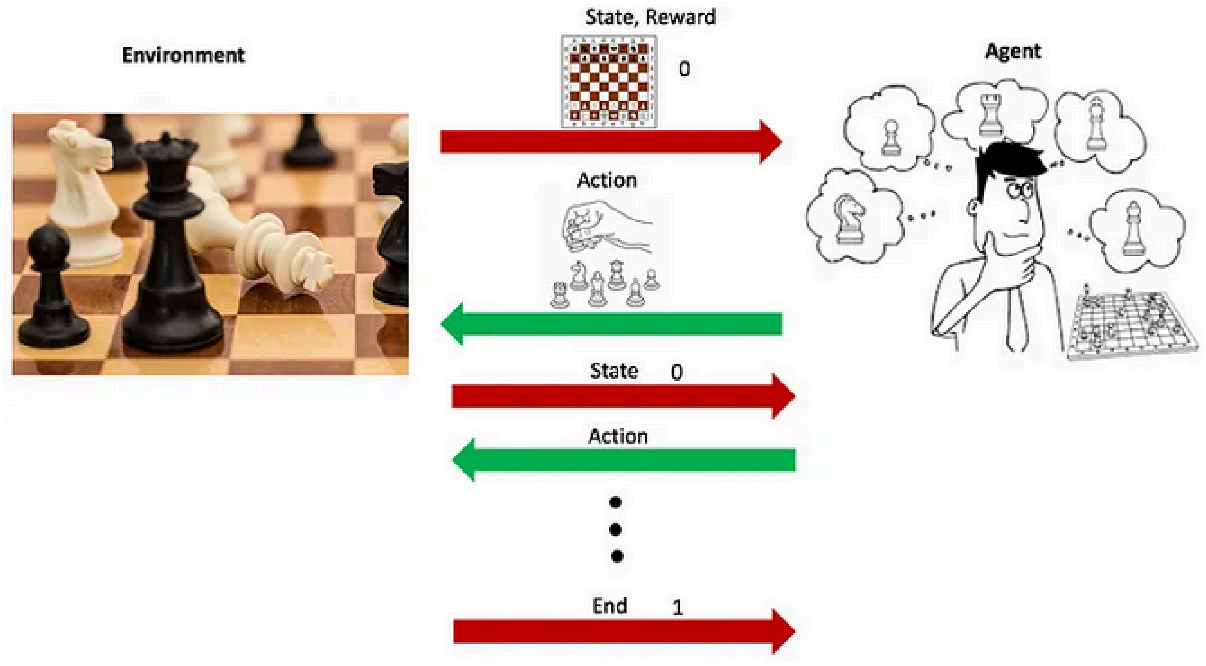
To better understand the interplay between chess and reinforcement learning, let’s look at some visual examples:

**Chessboard State Representation:**

In this image, the pieces are set up in their initial positions. Each piece's movement capabilities and strategic value play a critical role in the decision-making process of a chess engine.

**Reinforcement Learning Workflow:**

This diagram illustrates the basic workflow of reinforcement learning. The agent (chess engine) interacts with the environment (chessboard), makes moves (actions), and receives feedback (rewards), which it uses to learn and improve its strategy over time.



**Monte Carlo Tree Search (MCTS):**

MCTS is a popular algorithm used in reinforcement learning for game playing. It simulates many possible future game states to make informed decisions, balancing exploration of new moves and exploitation of known successful strategies.

## Research papers findings

**1.** Giraffe: Using Deep Reinforcement Learning to Play Chess

Key technique used in the implementation of chess engines that could be relevant for reinforcement learning applications:

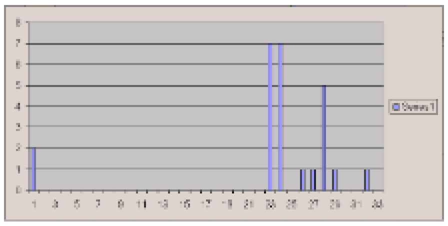
1. **Alpha-Beta Pruning**:
   - **Description**: Alpha-beta pruning is an optimization technique used in the minimax algorithm. It reduces the number of nodes evaluated in the search tree by eliminating branches that cannot possibly influence the final decision. This is done by maintaining two values, alpha and beta, which represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of, respectively.
   - **Relevance to Reinforcement Learning**: In reinforcement learning, particularly in environments with large action spaces and strategic depth like chess, efficiently exploring the decision space is crucial. Alpha-beta pruning helps in narrowing down the search to the most promising actions, thus speeding up learning by

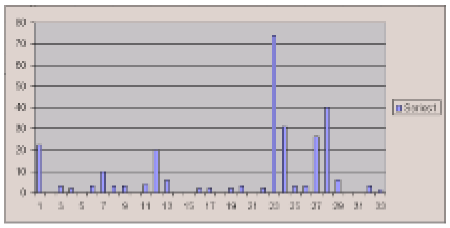
focusing computational resources on evaluating more relevant outcomes. This concept can be applied to reinforcement learning algorithms that need to handle large decision trees, helping to prioritize exploration and exploitation effectively.

Alpha-beta pruning enhance the efficiency and effectiveness of decision-making processes in chess engines. When applied to reinforcement learning, these techniques can significantly improve the learning algorithms' ability to make informed decisions in complex environments.

## 2. Using Reinforcement Learning in Chess Engines

Some insights:


(a)


(b)

In the context of using reinforcement learning (RL) for chess engines, two innovative techniques are highlighted in the document:

1. **Temporal Difference (TD) Learning**:
    - **TD Learning in Chess**: This technique involves the application of TD learning, a type of reinforcement learning that adjusts the value of a state based on the difference between the predicted value of the next state and the current state. This method is instrumental in refining the evaluation function of a chess program, helping it to learn from each game it plays by adjusting the coefficients used in these functions.
    - **Optimization with TD-Leaf**: The adaptation known as TD-Leaf optimizes this approach further by applying the TD updates not just on root nodes but throughout the best leaf nodes in the search tree. This method allows the chess engine to consider more in-depth game scenarios, improving the precision of its learning process and enabling more strategic insights into possible future moves.
2. **Machine Learning-Assisted Move Evaluation and Ordering**:
    - **Enhanced Move Ordering**: To improve the efficiency of the chess engine's search algorithm, machine learning techniques are used to predict and prioritize the most promising moves. By learning from past games, the engine can better order the moves to explore, using historical data to estimate which moves are likely to lead to favorable outcomes.

- **Classification of Board States**: This involves using a detailed database of board states to train the chess engine, allowing it to classify and evaluate different game scenarios more accurately. By adjusting the coefficients for each class of board state individually, the engine can tailor its strategy to specific situations, thereby enhancing its performance and adaptability in varied chess matches.

These techniques represent a significant shift from traditional brute-force methods to more nuanced, learning-driven approaches, allowing chess programs to improve their performance dynamically by learning from each game and adjusting their strategy accordingly.

## 3. Reinforcement Learning in an AdaptableChess Environment for DetectingHuman-understandable Concepts

The paper discusses the use of Reinforcement Learning (RL) combined with a concept detection method in a chess environment to make the deep neural network's decision-making process more interpretable. Here's a concise explanation of one technique used:

**Reinforcement Learning with Self-Play**: The chess agents are trained through self-play using reinforcement learning. This means that the agents play games against versions of themselves without any prior knowledge or strategy pre-installed. They start from scratch with randomly initialized weights in their neural networks and learn solely from the experience gained through gameplay. This approach allows the agents to explore a wide range of strategies and tactics, learning effective moves and strategies by continuously updating their neural network models based on the outcomes of each game. The agents optimize their play by trying to maximize their expected reward, which in the context of chess, involves winning the game or achieving a draw in disadvantageous situations.

This technique leverages the power of simulation and machine learning to enable a computer program to improve autonomously by learning from its own actions and experiences, without human intervention or detailed programming regarding the specifics of the chess strategy.

## 4. Investigations into Playing Chess Endgames using Reinforcement Learning.

Some insights:

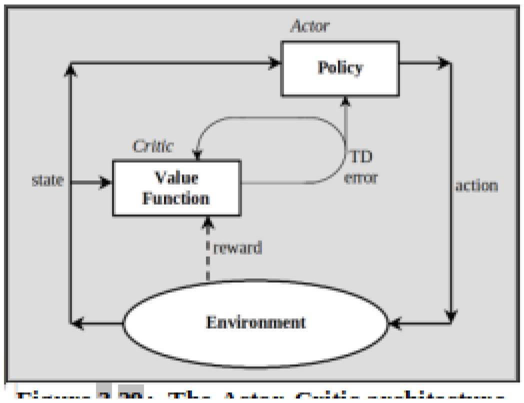


Figure 2.20: The Actor-Critic architecture

The document discusses two key techniques used in the implementation of chess using reinforcement learning:

1. **Temporal Difference (TD) Learning**: This method is a cornerstone of reinforcement learning for chess, where the agent learns from the experience by estimating the value functions directly from the actual rewards and without requiring a model of the environment. TD Learning updates the value estimates based on the differences between consecutive predictions, adjusting them closer to the actual received rewards. This allows the learning agent to effectively adjust its strategy incrementally, improving its performance over time through self-play and exploration of new strategies.
2. **Monte Carlo (MC) Methods**: Unlike TD Learning, Monte Carlo methods require the completion of an entire episode (or game) to update the policy. Here, the learning agent's understanding is adjusted post-game, based on the outcomes and the sequence of states and decisions taken during the game. This method averages the returns following all visits to a particular state, which then directly informs the policy improvements. In chess, this means analyzing entire games from start to finish and updating the strategies based on the wins or losses, making it well-suited for episodic tasks like individual chess games where the feedback is clear and terminal states (end of the game) are well-defined.

Both techniques harness the agent's interactions within the game environment to learn from actions' consequences without explicit teaching, simulating a trial-and-error learning process akin to natural learning processes observed in humans and animals.

## 5. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm

The "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm" by David Silver et al. introduces two fundamental techniques for implementing chess using reinforcement learning within the AlphaZero algorithm:

1. **Monte Carlo Tree Search (MCTS)**: AlphaZero employs MCTS instead of the traditional alpha-beta pruning used in earlier chess engines like Stockfish. MCTS enhances the

exploration of the game space by randomly simulating games from the current position to terminal states. Through repeated simulations, it builds a tree of game states and uses the results of these simulations to estimate the value of moves. This method helps in balancing between exploring new moves and exploiting known advantageous moves, thereby enabling effective decision-making without prior game-specific knowledge.

2. **Deep Neural Networks for Policy and Value Estimation**: Unlike traditional chess programs that rely on handcrafted evaluation functions developed by human experts, AlphaZero uses a deep neural network to learn both move probabilities and position value estimates from self-play. The neural network outputs a probability distribution over possible moves (policy) and an estimate of the expected outcome from current positions (value). This approach allows the system to learn successful strategies and tactics purely from the game's rules and its experience gained through playing against itself, starting from random play.

Together, these techniques allow AlphaZero not only to master the game of chess but also to generalize across other complex games like Shogi and Go, achieving superhuman performance by learning from scratch without relying on historical data or human expertise.

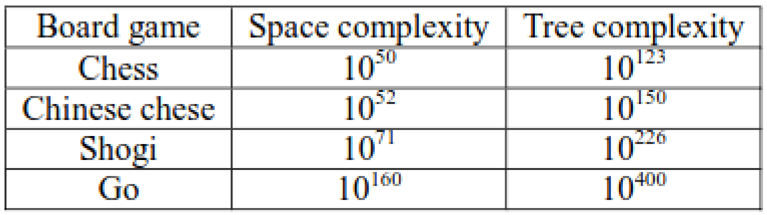
## 6. IMPLEMENTATION OF A CHESS GAME PLAYING SYSTEM USING MACHINE LEARNING

In the paper titled "Implementation of a Chess Game Playing System Using Machine Learning" by **Mr. Suhas Raut** and colleagues, the technique utilized for implementing chess involves the use of Convolutional Neural Networks (CNNs) in a supervised learning framework. This approach treats chess as a regression problem, where the system predicts the best move from a given board position. The chessboard is represented as an 8x8 matrix and modeled using bitboard representation, which encodes each piece and empty spaces into a binary format. This format enhances performance during data training by optimizing the input for convolutional processing. The CNN architecture employed consists of multiple convolutional layers followed by fully connected layers, using ReLU activation functions and the tanh function for the output layer, optimizing predictions via mean squared error loss and an Adam optimizer. This setup allows the system to learn chess strategies and tactics directly from game data, enabling it to recognize patterns and predict optimal moves without manual feature engineering or explicit strategy programming.

## 7. Application of Q-Learning and RBF Network in Chinese Chess Game System

Some insights from paper:

**Table 1.** Spatial Complexity and Tree Complexity of Several Kinds of Chess.



| Board game | Space complexity | Tree complexity |
|---|---|---|
| Chess | $10^{50}$ | $10^{123}$ |
| Chinese chese | $10^{52}$ | $10^{150}$ |
| Shogi | $10^{71}$ | $10^{226}$ |
| Go | $10^{160}$ | $10^{400}$ |

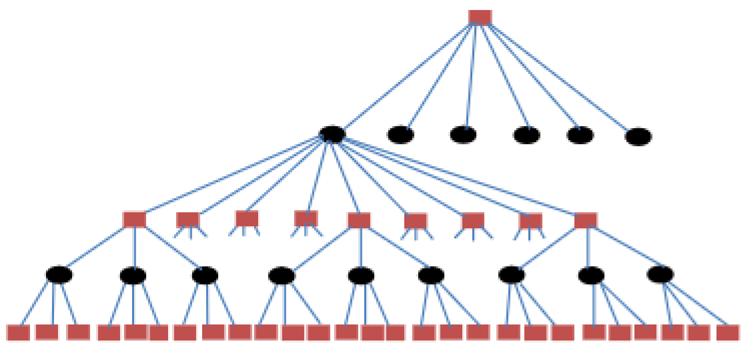


**Figure 1.** Four-tier Game Tree.

In the article "Application of Q-Learning and RBF Network in Chinese Chess Game System" by **Wenyang He et al**., the Q-learning technique is employed to enhance the learning and strategic decision-making capabilities of a Chinese Chess game system. Q-learning is a type of reinforcement learning that helps the system learn optimal actions based on trial and error, without needing a model of the environment. This method involves updating the Q-values (quality of actions) associated with each possible state and action in the game. The system iteratively improves its policy by exploring actions and updating the Q-values based on the rewards received after performing those actions. This exploration is balanced with exploitation (choosing the best-known action) using an ε-greedy strategy, where ε is a parameter that controls the trade-off between exploring new actions and exploiting known valuable actions. The learning process allows the system to make increasingly informed decisions, thereby enhancing its ability to play Chinese Chess effectively.

## 8. AlphaGo Zero & AlphaZero Mastering Go, Chess and Shogi wihtout human knowledge

AlphaGo Zero and AlphaZero represent significant advancements in the use of reinforcement learning for mastering board games like Go, Chess, and Shogi, developed by Google DeepMind. Unlike their predecessors, these programs do not rely on historical game data or human expertise; instead, they learn from scratch using a method called self-play, where the system plays against itself to improve.

**AlphaGo Zero** combines a deep neural network with a Monte Carlo Tree Search (MCTS) algorithm. The neural network is trained to predict both game moves and the winner's identity, using outcomes of games it plays against itself. This approach simplifies previous models by using a single network for both the policy (suggesting moves) and value (estimating probabilities of winning). The system progressively refines its predictions and strategies, achieving superhuman capabilities without human-derived data.

**AlphaZero** extends the methodology of AlphaGo Zero to other games such as Chess and Shogi. It uses a similar self-play reinforcement learning framework but adapts to the unique aspects of these games. For instance, AlphaZero does not use data augmentation because Chess and Shogi lack the symmetrical properties of Go. Also, it considers potential outcomes like draws, which are particularly relevant in Chess and Shogi. AlphaZero operates under the same principles, using a generalized version of the algorithm without game-specific enhancements, demonstrating its ability to excel across different board games using a uniform approach.

Both systems showcase the power of combining deep learning with reinforcement learning, allowing the programs to discover novel strategies and insights that go beyond traditional human knowledge, fundamentally changing the approach to AI in game theory and decision-making processes.

## 9. A general reinforcement learningalgorithm that masters chess, shogi,and Go through self-play

The AlphaZero algorithm by David Silver and colleagues represents a significant breakthrough in reinforcement learning by achieving superhuman performance in chess, shogi, and Go without any domain-specific human knowledge. Unlike traditional chess engines that rely on sophisticated search techniques, domain-specific adaptations, and handcrafted evaluation functions, AlphaZero learns solely from self-play. Starting from a state of complete ignorance and only the rules of the game as input, it uses a deep neural network to evaluate board positions and decide on moves.

AlphaZero employs a Monte Carlo Tree Search (MCTS) that is guided by the neural network. The network predicts move probabilities and the game outcome from given positions, learning from games it plays against itself. This approach allows it to optimize its playing strategy continually through a process of trial and error, refining its predictions and moves based on the outcomes of previous games.

## 10. AlphaZero Vs StockFish – A Review of Chess Engines

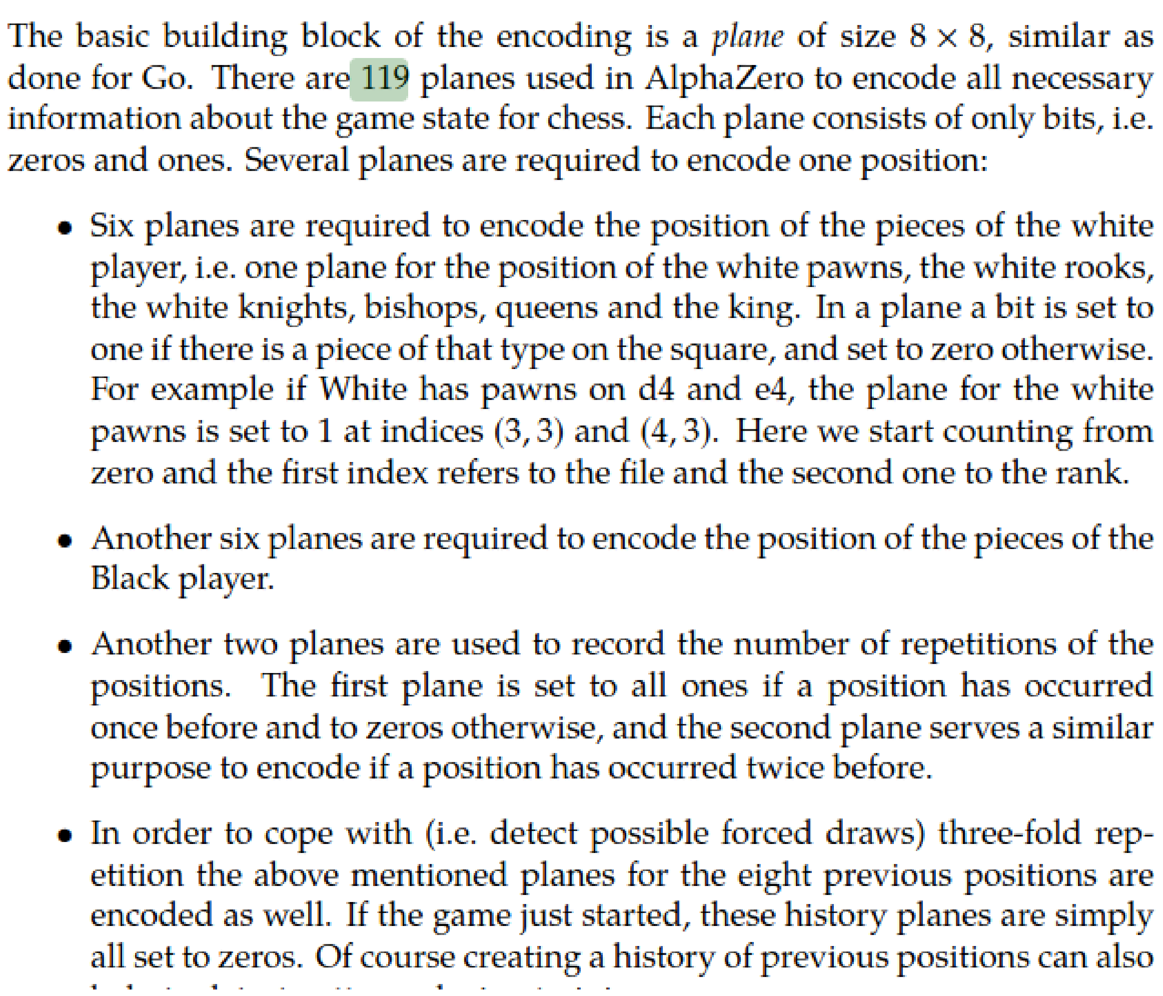
Stockfish employs traditional AI methods using alpha-beta pruning and extensive databases of past games, excelling in chess by analyzing millions of potential moves rapidly. It uses domain-specific knowledge, including handcrafted evaluation functions, to optimize its play strategy.

AlphaZero, developed by DeepMind, adopts a novel approach through reinforcement learning and deep neural networks. It learns chess from scratch, enhancing its skills via a self-play system that refines its strategies over time without preloaded game knowledge. It utilizes Monte Carlo Tree Search to explore and evaluate moves based on outcomes from its neural network, focusing on quality over quantity of move exploration.

While Stockfish relies on detailed, pre-programmed knowledge and raw computational power, AlphaZero's method is more adaptive, emphasizing continual learning and strategic flexibility derived from fundamental game rules.

## 11. Neural Networks for Chess

This is the book we mainly referred to for implementing Alpha Zero. This book guided us through specifications of implementation which were very difficult to be found. Like having 119 input features but how to design them. In this case , book guided us as:



The basic building block of the encoding is a *plane* of size $8 \times 8$, similar as done for Go. There are 119 planes used in AlphaZero to encode all necessary information about the game state for chess. Each plane consists of only bits, i.e. zeros and ones. Several planes are required to encode one position:

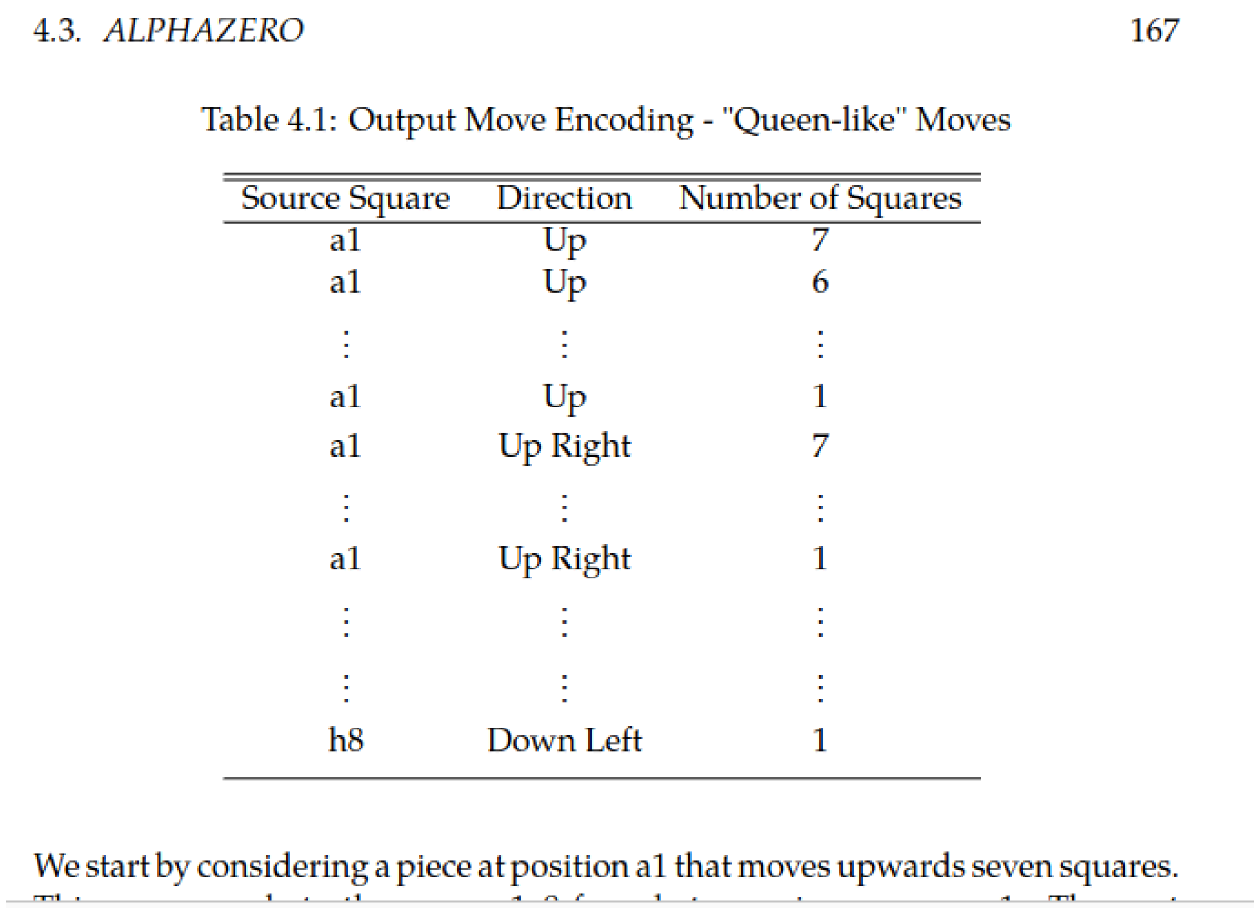
- Six planes are required to encode the position of the pieces of the white player, i.e. one plane for the position of the white pawns, the white rooks, the white knights, bishops, queens and the king. In a plane a bit is set to one if there is a piece of that type on the square, and set to zero otherwise. For example if White has pawns on d4 and e4, the plane for the white pawns is set to 1 at indices $(3,3)$ and $(4,3)$. Here we start counting from zero and the first index refers to the file and the second one to the rank.

- Another six planes are required to encode the position of the pieces of the Black player.

- Another two planes are used to record the number of repetitions of the positions. The first plane is set to all ones if a position has occurred once before and to zeros otherwise, and the second plane serves a similar purpose to encode if a position has occurred twice before.

- In order to cope with (i.e. detect possible forced draws) three-fold repetition the above mentioned planes for the eight previous positions are encoded as well. If the game just started, these history planes are simply all set to zeros. Of course creating a history of previous positions can also

The book also guided us about different other techniques and then we chose alpha zero finally. Other example of book giving us information about all possible chess moves outputs( which is a very serious problem) is



4.3. *ALPHAZERO* 167

Table 4.1: Output Move Encoding - "Queen-like" Moves

| Source Square | Direction | Number of Squares |
|---|---|---|
| a1 | Up | 7 |
| a1 | Up | 6 |
| ⋮ | ⋮ | ⋮ |
| a1 | Up | 1 |
| a1 | Up Right | 7 |
| ⋮ | ⋮ | ⋮ |
| a1 | Up Right | 1 |
| ⋮ | ⋮ | ⋮ |
| ⋮ | ⋮ | ⋮ |
| h8 | Down Left | 1 |

We start by considering a piece at position a1 that moves upwards seven squares.

This is the source where we got most of stuff for coding Alpha zero for chess.

## Summary of the Techniques in the Research Papers:

1. ***Alpha-Beta Pruning:***
   - **Advantages**: Reduces the number of nodes evaluated in the decision-making process, optimizing the speed and efficiency of game tree exploration.
   - **Flaws**: While it cuts down unnecessary paths, it may overlook potential strategic depth if not properly calibrated.
2. ***Q-Learning:***

   **Advantages:**

- **Flexibility:** Q-learning is model-free, meaning it does not require knowledge of the environment's dynamics and can learn optimal actions by trial and error directly from interactions with the environment.
- **Off-Policy Learning:** It can learn the optimal policy independently of the agent's actions, allowing it to effectively utilize off-policy data and improve from exploratory actions.

**Flaws:**

- **Slow Convergence:** Q-learning can take a long time to converge to an optimal policy, especially in environments with large state or action spaces, due to the need to estimate the value of each action at each state.
- **Overestimation Bias:** It tends to overestimate action values because it uses the maximum expected reward for updates, which can lead to suboptimal policy learning if not managed properly.

3. ***Temporal Difference (TD) Learning:***
   - **Advantages**: Allows systems to learn optimal strategies from the final outcome of sequences, adjusting strategies incrementally.
   - **Flaws**: Relies heavily on the quality of the sequence of game states, and poor sequences can lead to slower or suboptimal learning.
4. ***Monte Carlo Tree Search (MCTS):***
   - **Advantages**: Enhances exploration of the game space without needing extensive databases, allowing discovery of new strategies.
   - **Flaws**: Computationally intensive as it requires many random simulations to statistically estimate the best moves.
5. ***Deep Neural Networks for Policy and Value Estimation:***
   - **Advantages**: Learns complex patterns and strategies directly from gameplay, adjusting to new situations dynamically.
   - **Flaws**: Requires substantial computational resources for training; initial training phases may involve many suboptimal plays before achieving proficiency.

## Choice of AlphaZero:

Despite the presence of traditional and robust methods like alpha-beta pruning used by engines like Stockfish, AlphaZero was chosen due to several compelling reasons:

AlphaZero stands out compared to other algorithms like Stockfish, Q-learning, TD learning, AlphaGo Zero, and AlphaGo due to several key advantages:

**Compared to Stockfish:**

- **Flexibility:** Unlike Stockfish, which relies heavily on pre-programmed opening books and endgame tablebases, AlphaZero learns entirely from self-play, without any reliance on prior human knowledge. This allows AlphaZero to discover novel strategies and adapt dynamically to the opponent’s play style.

- **Generalization:** AlphaZero uses a generalized approach that can be applied to multiple games (chess, shogi, and Go), showing its robustness and versatility compared to Stockfish's chess-specific programming.

**Compared to Q-learning and TD learning:**

- **Efficiency in Complex Environments:** AlphaZero combines deep neural networks with Monte Carlo Tree Search (MCTS), effectively handling the vast state spaces of complex board games better than standard Q-learning and TD learning, which often struggle with large or continuous action spaces without extensive modification or feature engineering.
- **Balanced Exploration and Exploitation:** AlphaZero's use of MCTS naturally balances exploration and exploitation, using its neural network to guide the search more intelligently than typical epsilon-greedy strategies in Q-learning.

**Compared to AlphaGo and AlphaGo Zero:**

- **Self-Sufficiency:** Unlike AlphaGo, which required vast amounts of human gameplay data for training, AlphaZero learns entirely from scratch, using only the rules of the game. This self-sufficiency leads to the development of creative and unconventional strategies that might not be found in human play.
- **Rapid Learning:** AlphaZero achieved superior performance in a shorter training time compared to AlphaGo Zero, demonstrating faster learning capabilities and more efficient processing.

Overall, AlphaZero represents a significant step forward in the use of AI for complex decision-making tasks, demonstrating the potential of reinforcement learning and neural networks to surpass traditional methods in both performance and strategic innovation.

# Current bots

Our main focus of study was two of the most popular chess bots that are currently used almost everywhere in the chess world.

- Stockfish
- AlphaZero

Below we discuss an overview of how both of these bots were trained and how they reached the level that they are at.

**Stockfish:**

Stockfish is a relatively older bot than AlphaZero and has been around for about 16 years. It has been one of the best chess engines in the world for several years; it has won all main events of the Top Chess Engine Championship (TCEC) and the Chess.com Computer Chess Championship (CCC) since 2020 and, as of May 2024, is the strongest CPU chess engine in the world with an estimated Elo rating of 3634.

**Algorithm used to train:**

Stockfish uses a calculation method known as minimax. In the minimax algorithm, Stockfish considers a set of potential strategies (current moves which can be played) and the best strategies that can be utilized by the opponent as a response to each strategy.

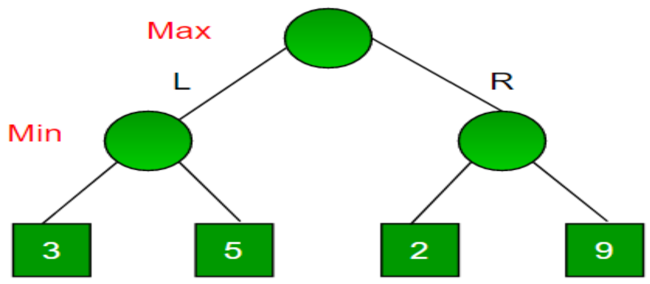
**Minimax:**

Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn-based games such as Tic-Tac-Toe, Backgammon, Mancala, Chess, etc.

In Minimax the two players are called maximizer and minimizer. The maximizer tries to get the highest score possible while the minimizer tries to do the opposite and get the lowest score possible.

Every board state has a value associated with it. In a given state if the maximizer has the upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.

**Example**



Being the maximizer you would choose the larger value that is 3. Hence the optimal move for the maximizer is to go LEFT and the optimal value is 3.

**AlphaZero:**

AlphaGo was a bot that played the game Go that took the world by storm. However, some time later, the same people who made this bot made another one named AlphaZero. This bot, however, was very different from the previous one as it could learn to play any board game similar to Go, like Chess and Shogi. Not only that, but this bot beat AlphaGo in Go, which it was specifically made for. Now back to chess. This bot beat Stockfish 8 (Stockfish 16.1 is the latest which is better than AlphaZero), which was the best bot for many years, after training for only four hours.

**Algorithm used to train:**

The game-playing agent design that AlphaZero uses is based on a version of Monte Carlo Tree Search, which, instead of exploring every possible branch as deeply as the hardware allows like DFS does, explores a small number of branches but until the very end of the game. The basic premise behind MCTS is that it does not require exploring as many branches as DFS, and the extra computational power can be redirected elsewhere.

**Monte Carlo Tree Search:**

Monte Carlo Tree Search is an algorithm very similar to the minimax algorithm discussed above. It is a method for making optimal decisions in artificial intelligence (AI) problems, typically move planning in combinatorial games. It combines the generality of random simulation with the precision of tree search.

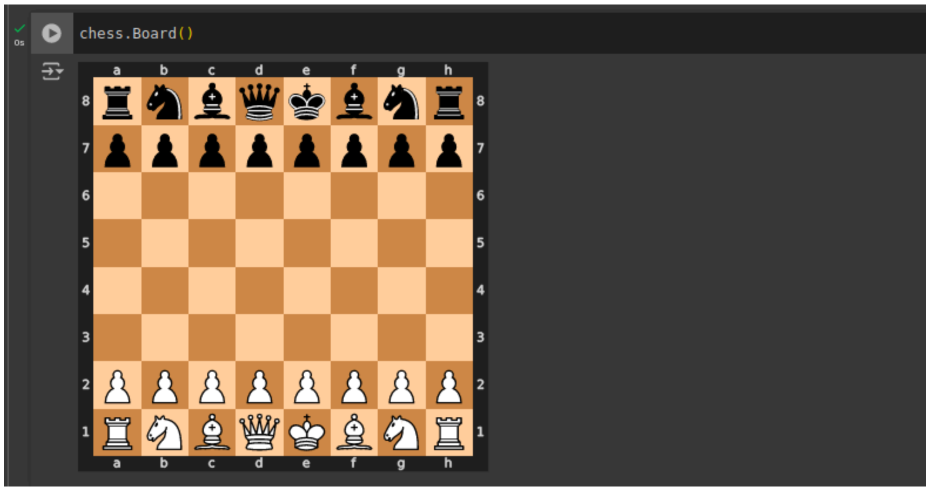
The final tree is seen the same way as a minimax tree is seen. However, the process of making the tree is different from minimax.

# <u>Our Implementation</u>

## <u>Environment</u>

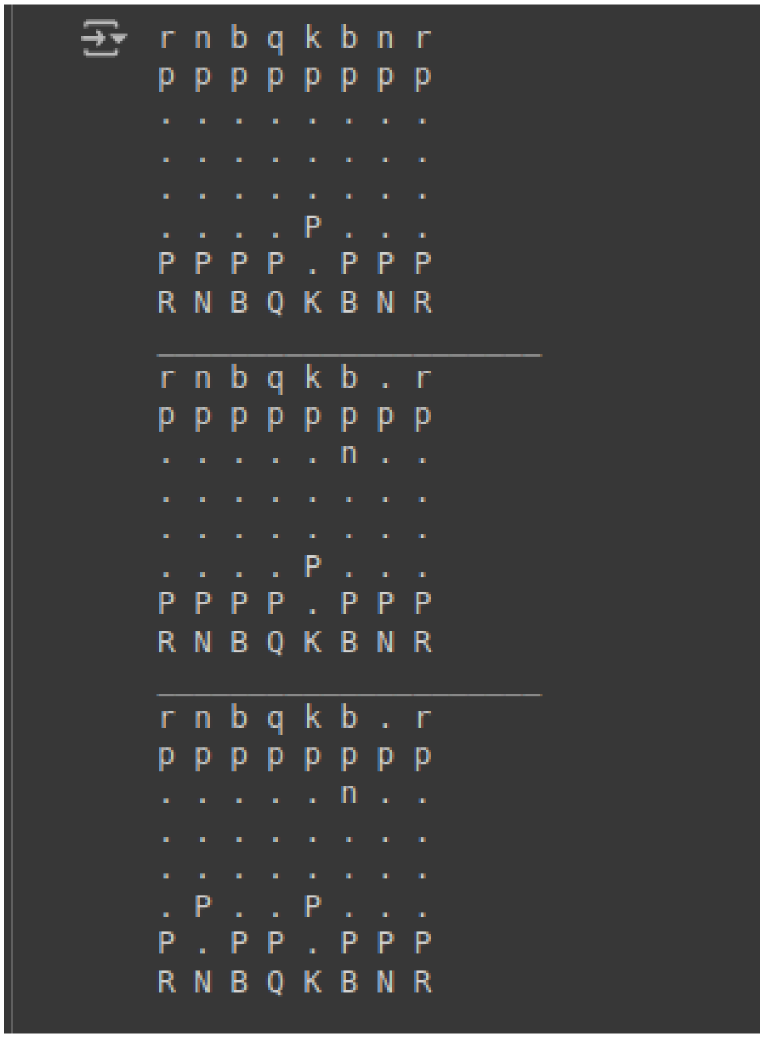
First and foremost we will talk about the environment we used to actually display and play the games of chess. We used a library called **python-chess** which is a chess library for Python, with move generation, move validation, and support for common formats.
Its features include:

- Creating a Board object to play a game on
- Rendering of an svg in ipynb notebooks for a Board object



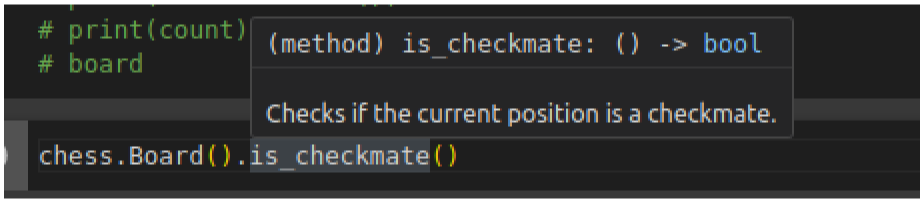
```
chess.Board()
```

- Show a simple ASCII board for normal print statements



```
r n b q k b n r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . P . . .
P P P P . P P P
R N B Q K B N R
________________
r n b q k b . r
p p p p p p p p
. . . . . n . .
. . . . . . . .
. . . . . . . .
. . . . P . . .
P P P P . P P P
R N B Q K B N R
________________
r n b q k b . r
p p p p p p p p
. . . . . n . .
. . . . . . . .
. . . . . . . .
. P . . P . . .
P . P P . P P P
R N B Q K B N R
```

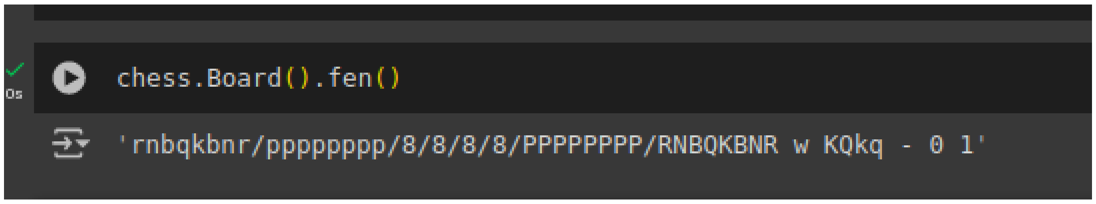
- Has functions to detect checkmates, stalemates and draws by insufficient material from the state of the current Board.



```
# print(count)
# board

chess.Board().is_checkmate()
```

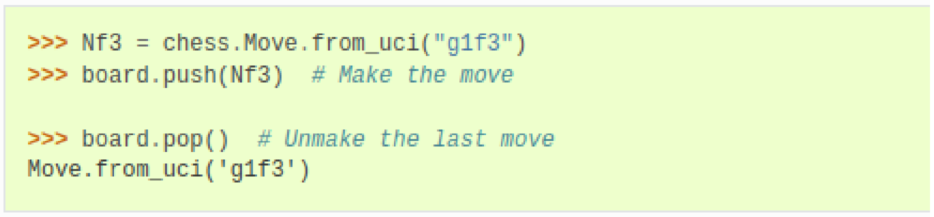
```
(method) is_checkmate: () -> bool

Checks if the current position is a checkmate.
```

- Parses and creates FENs (Forsyth-Edwards Notation, and it is the standard notation to describe positions of a chess game) for every board.



```
chess.Board().fen()
'rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1'
```

- Allows you to make and unmake moves on a board.



```
>>> Nf3 = chess.Move.from_uci("g1f3")
>>> board.push(Nf3)  # Make the move

>>> board.pop()  # Unmake the last move
Move.from_uci('g1f3')
```

**<u>Method of Training in this environment</u>**

As we discussed earlier, Stockfish uses minimax and AlphaZero uses Monte Carlo Tree Search. We decided to go with the approach AlphaZero used after all of our research and findings above.

However, our approach was a little different. This was due to several reasons, but the most significant one being that we did not have enough compute resources to train a model similar or in competition to AlphaZero.

Alpha Zero consisted of three main parts.

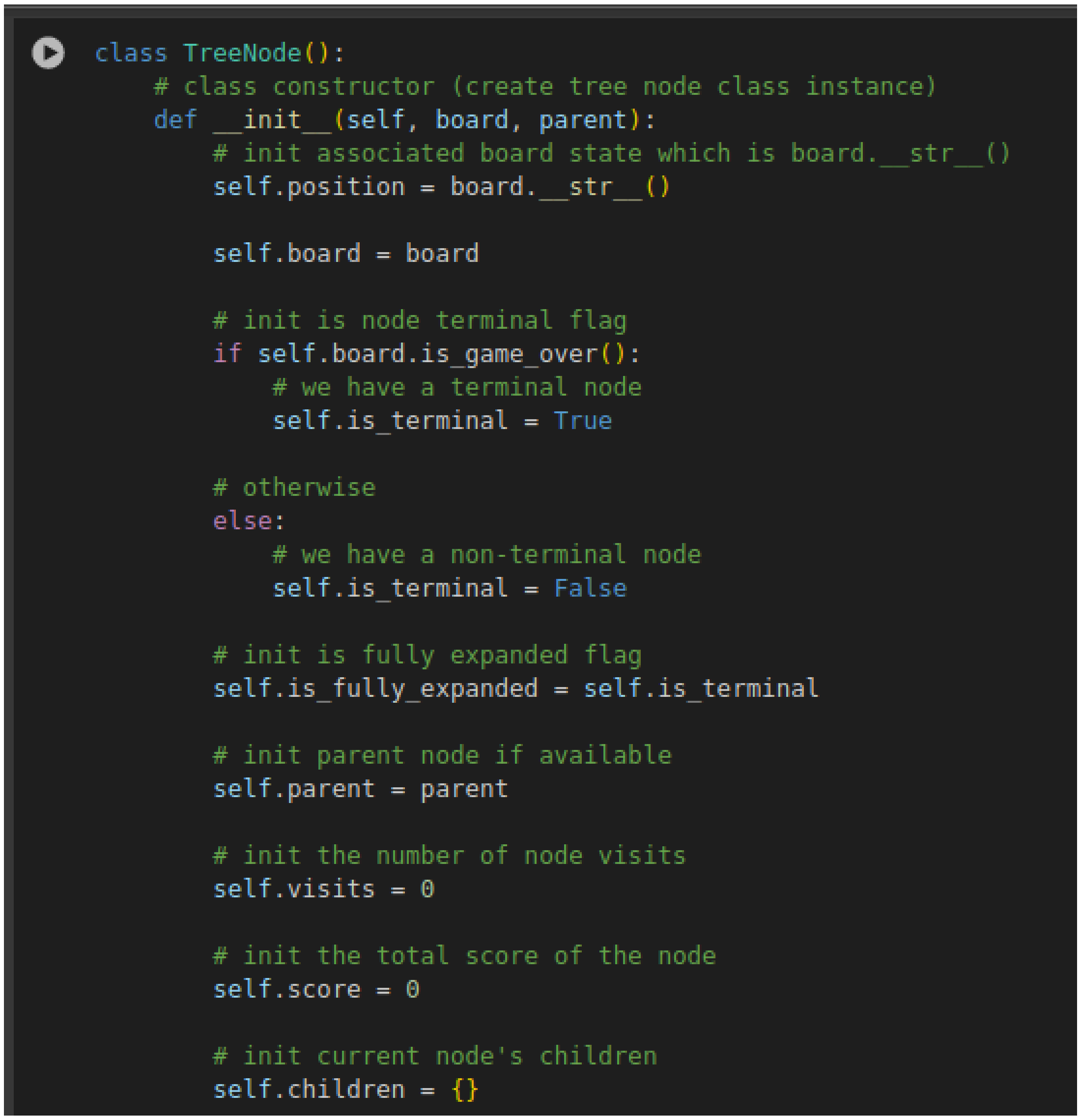
- A policy Network
- A value network
- Monte Carlo Tree Search

Now the way AlphaZero works is that it first trains the policy network and the value network on data. And then the reinforcement part starts when the data is now discarded and we have a trained policy and value network. This policy network and value network helps build the monte carlo tree and then this tree helps train the next iteration of the policy network. This policy network now plays a number of games against the previous policy network and if it wins more than 55% of the games, the policy network is updated to the newer one. And the cycle keeps repeating.

**First try at implementation:**

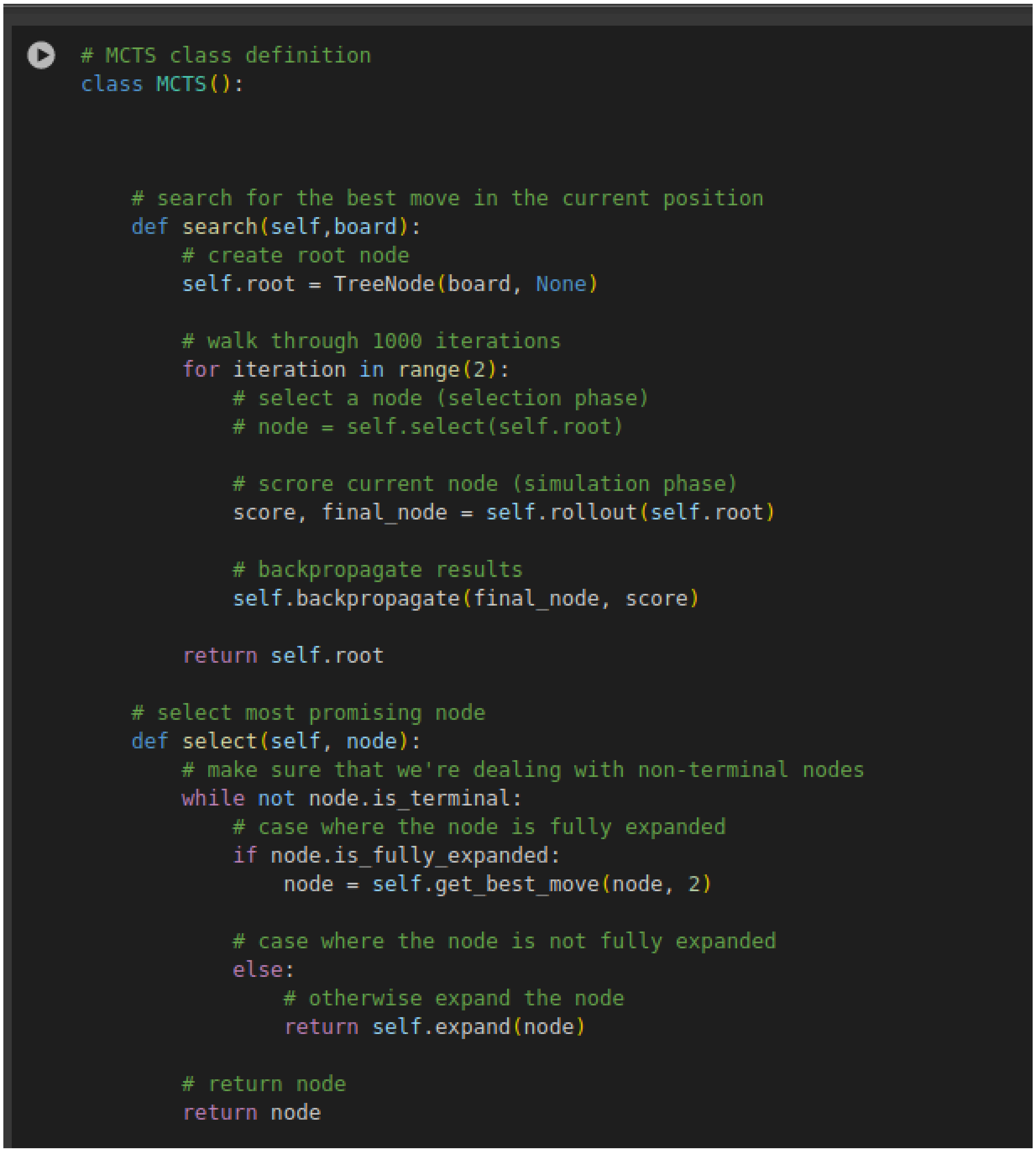
At first we tried this approach but we tried to do it completely from scratch. As in, we were trying to start without a policy network and tried to make a monte carlo tree by making random moves.

As to how we implemented the monte carlo tree, we created a class called TreeNode which represented a node in the tree, and had all the data about the state of the board in that node. The screenshot of the code is shown below:

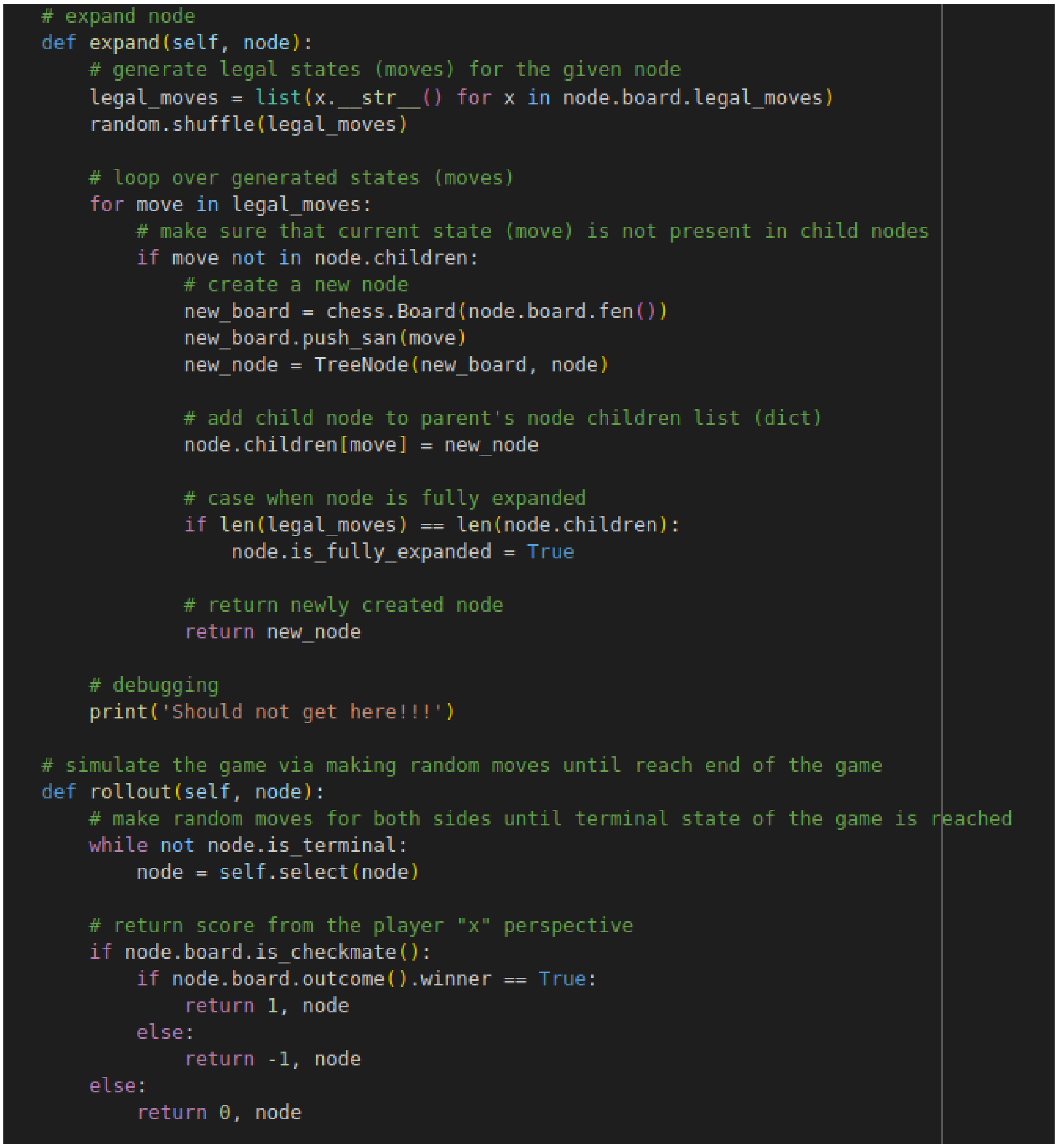


```python
class TreeNode():
    # class constructor (create tree node class instance)
    def __init__(self, board, parent):
        # init associated board state which is board.__str__()
        self.position = board.__str__()

        self.board = board

        # init is node terminal flag
        if self.board.is_game_over():
            # we have a terminal node
            self.is_terminal = True

        # otherwise
        else:
            # we have a non-terminal node
            self.is_terminal = False

        # init is fully expanded flag
        self.is_fully_expanded = self.is_terminal

        # init parent node if available
        self.parent = parent

        # init the number of node visits
        self.visits = 0

        # init the total score of the node
        self.score = 0

        # init current node's children
        self.children = {}
```

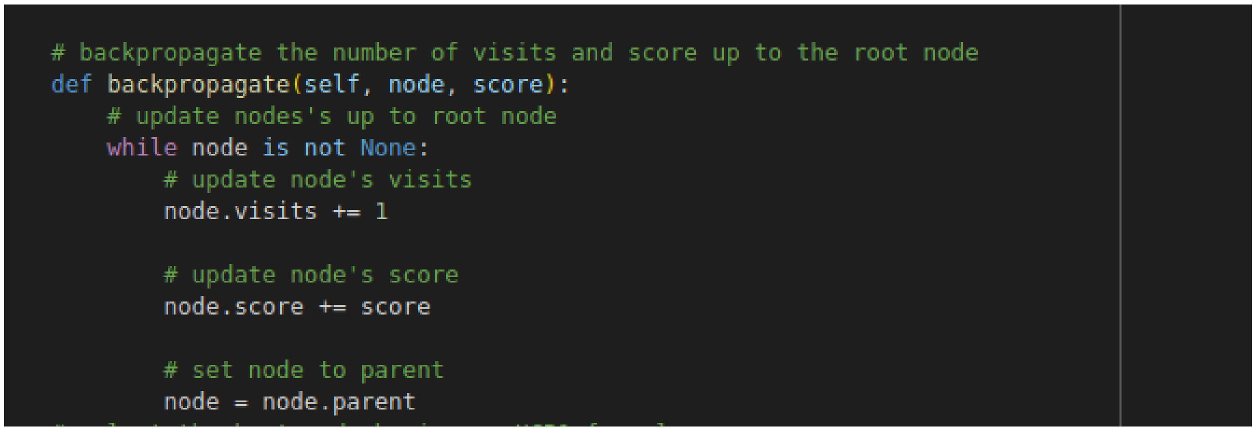
Now once we had the nodes, we wrote functions in another class to create the whole tree with connected nodes through random moves and assign scores to each node through backpropagation after ending each game. We also wrote a function that calculated the best move according to the current data of the tree which depended on the number of visits and the score of a child node so that the same node does not keep getting visited. The code is shown below:



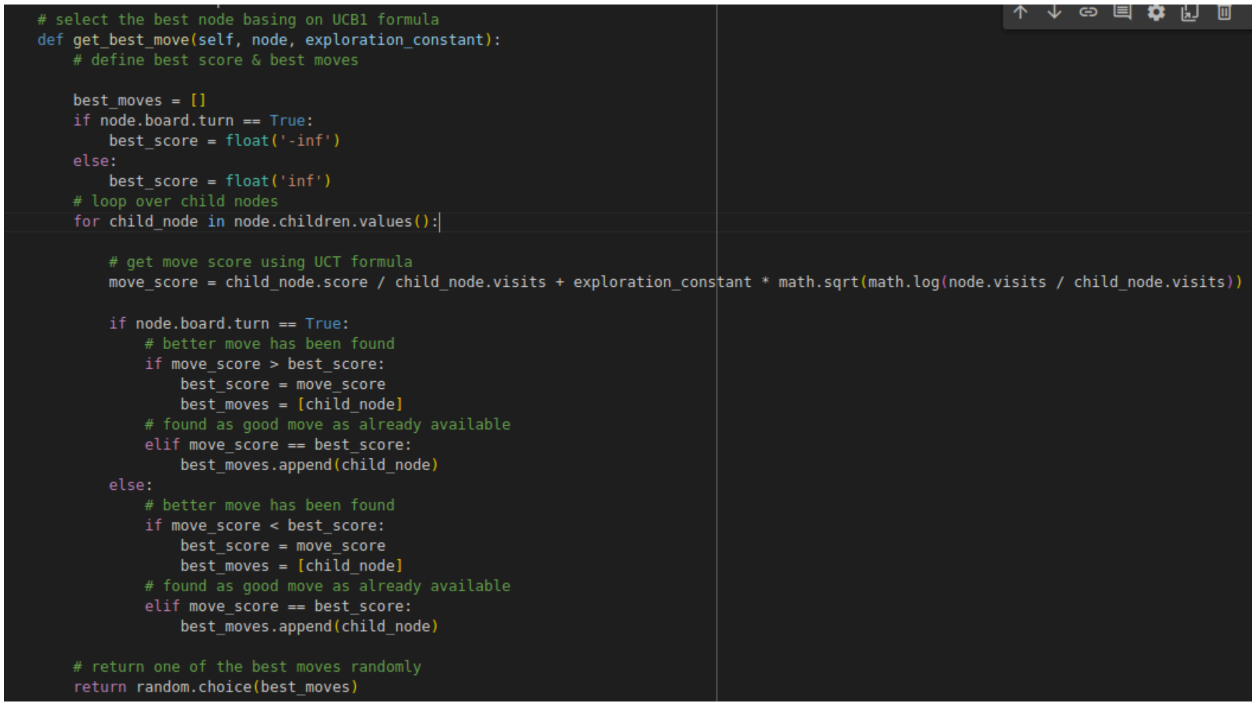
```
# MCTS class definition
class MCTS():



    # search for the best move in the current position
    def search(self,board):
        # create root node
        self.root = TreeNode(board, None)

        # walk through 1000 iterations
        for iteration in range(2):
            # select a node (selection phase)
            # node = self.select(self.root)

            # scrore current node (simulation phase)
            score, final_node = self.rollout(self.root)

            # backpropagate results
            self.backpropagate(final_node, score)

        return self.root

    # select most promising node
    def select(self, node):
        # make sure that we're dealing with non-terminal nodes
        while not node.is_terminal:
            # case where the node is fully expanded
            if node.is_fully_expanded:
                node = self.get_best_move(node, 2)

            # case where the node is not fully expanded
            else:
                # otherwise expand the node
                return self.expand(node)

        # return node
        return node
```



```python
    # expand node
    def expand(self, node):
        # generate legal states (moves) for the given node
        legal_moves = list(x.__str__() for x in node.board.legal_moves)
        random.shuffle(legal_moves)

        # loop over generated states (moves)
        for move in legal_moves:
            # make sure that current state (move) is not present in child nodes
            if move not in node.children:
                # create a new node
                new_board = chess.Board(node.board.fen())
                new_board.push_san(move)
                new_node = TreeNode(new_board, node)

                # add child node to parent's node children list (dict)
                node.children[move] = new_node

                # case when node is fully expanded
                if len(legal_moves) == len(node.children):
                    node.is_fully_expanded = True

                # return newly created node
                return new_node

        # debugging
        print('Should not get here!!!')

    # simulate the game via making random moves until reach end of the game
    def rollout(self, node):
        # make random moves for both sides until terminal state of the game is reached
        while not node.is_terminal:
            node = self.select(node)

        # return score from the player "x" perspective
        if node.board.is_checkmate():
            if node.board.outcome().winner == True:
                return 1, node
            else:
                return -1, node
        else:
            return 0, node
```



```
# backpropagate the number of visits and score up to the root node
def backpropagate(self, node, score):
    # update nodes's up to root node
    while node is not None:
        # update node's visits
        node.visits += 1

        # update node's score
        node.score += score

        # set node to parent
        node = node.parent
```



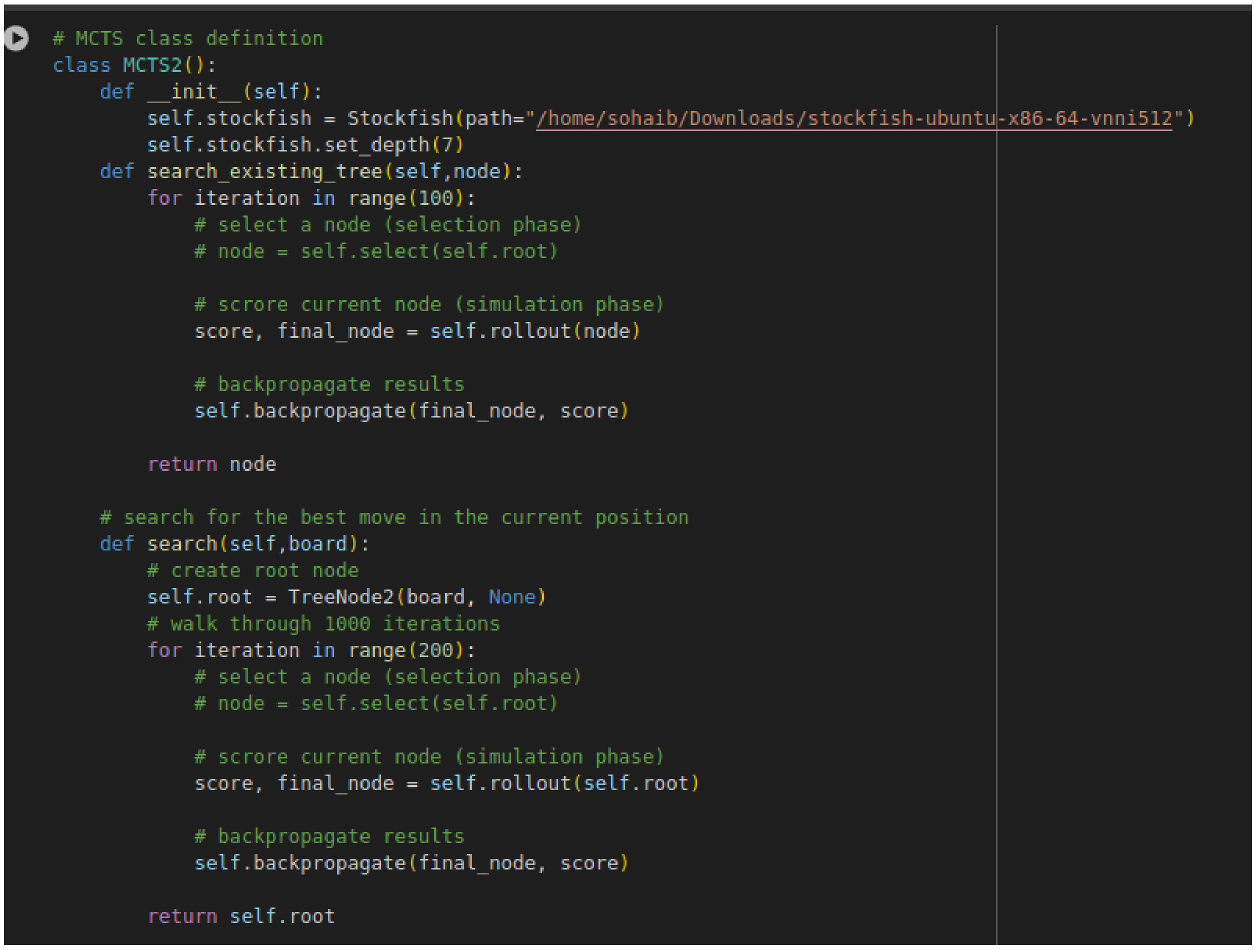
```
# select the best node basing on UCB1 formula
def get_best_move(self, node, exploration_constant):
    # define best score & best moves

    best_moves = []
    if node.board.turn == True:
        best_score = float('-inf')
    else:
        best_score = float('inf')
    # loop over child nodes
    for child_node in node.children.values():

        # get move score using UCT formula
        move_score = child_node.score / child_node.visits + exploration_constant * math.sqrt(math.log(node.visits / child_node.visits))

        if node.board.turn == True:
            # better move has been found
            if move_score > best_score:
                best_score = move_score
                best_moves = [child_node]
            # found as good move as already available
            elif move_score == best_score:
                best_moves.append(child_node)
        else:
            # better move has been found
            if move_score < best_score:
                best_score = move_score
                best_moves = [child_node]
            # found as good move as already available
            elif move_score == best_score:
                best_moves.append(child_node)

    # return one of the best moves randomly
    return random.choice(best_moves)
```

However, this whole implementation was useless. This is because we soon realized that the results were not getting better at all no matter how many iterations we ran, even though we had very limited compute resources, we could still see that this method was giving unpromising results. This was because we soon realized that the game of chess has too many possible moves to get a good result by playing random moves. It just trains the model to play randomly. Secondly, when the moves are random, winning a game of chess does not mean the moves you played were good.
Therefore, due to all these reasons we scratched the plan of making the bot from complete scratch.

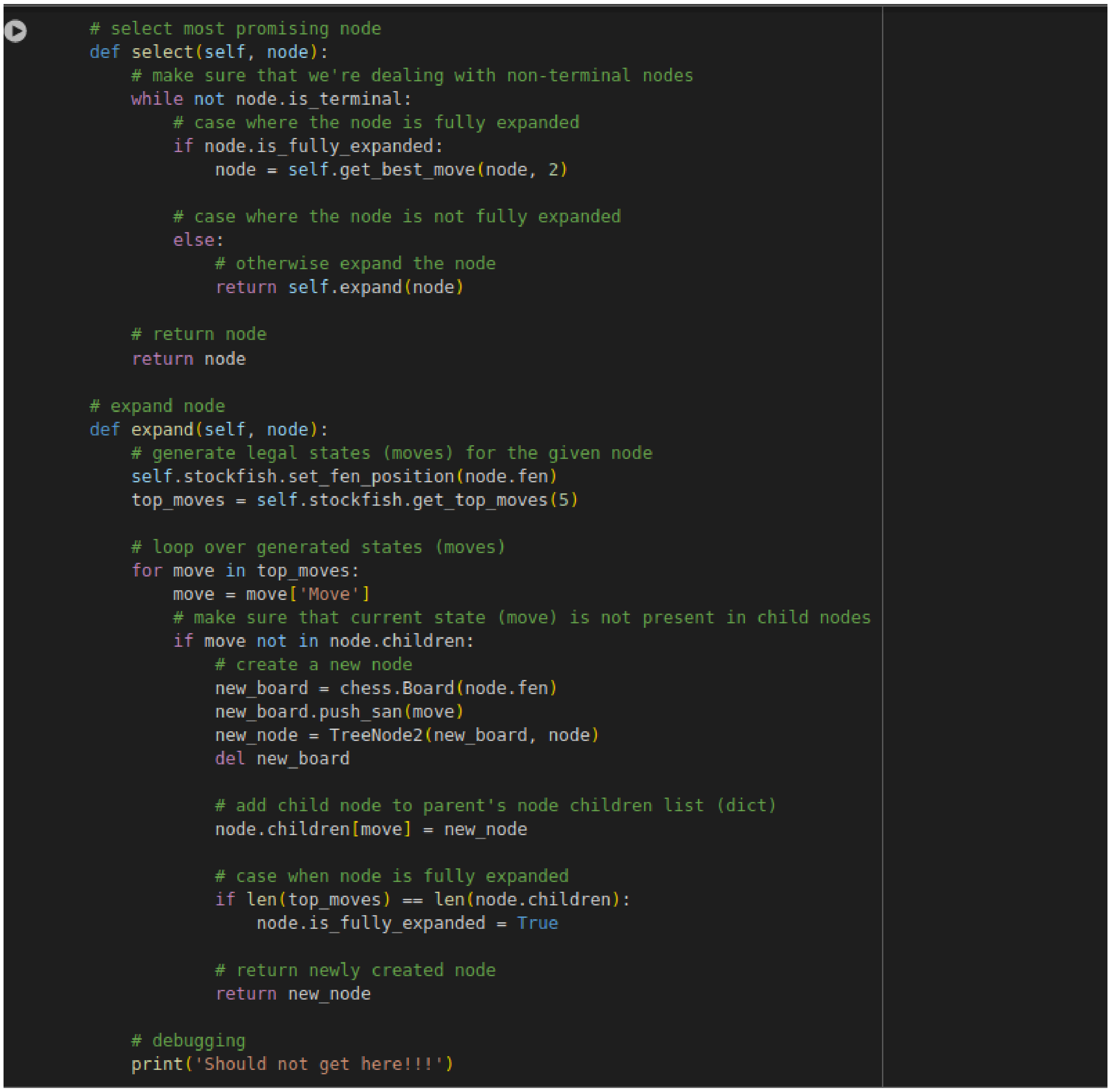
**Second try at implementation:**

Now in the second try, we changed our plan and decided to train the policy network first just like AlphaZero did. But instead of using datasets, we decided to use another bot, Stockfish, to help create a monte carlo tree which we would use to train our policy network. This was more of an instinctual decision as we believed that this would act like an already trained policy network just like the one AlphaZero used. We used stockfish at a very low-level so that we could get a decently trained model which we would then improve through reinforcement learning using the same method AlphaZero used.

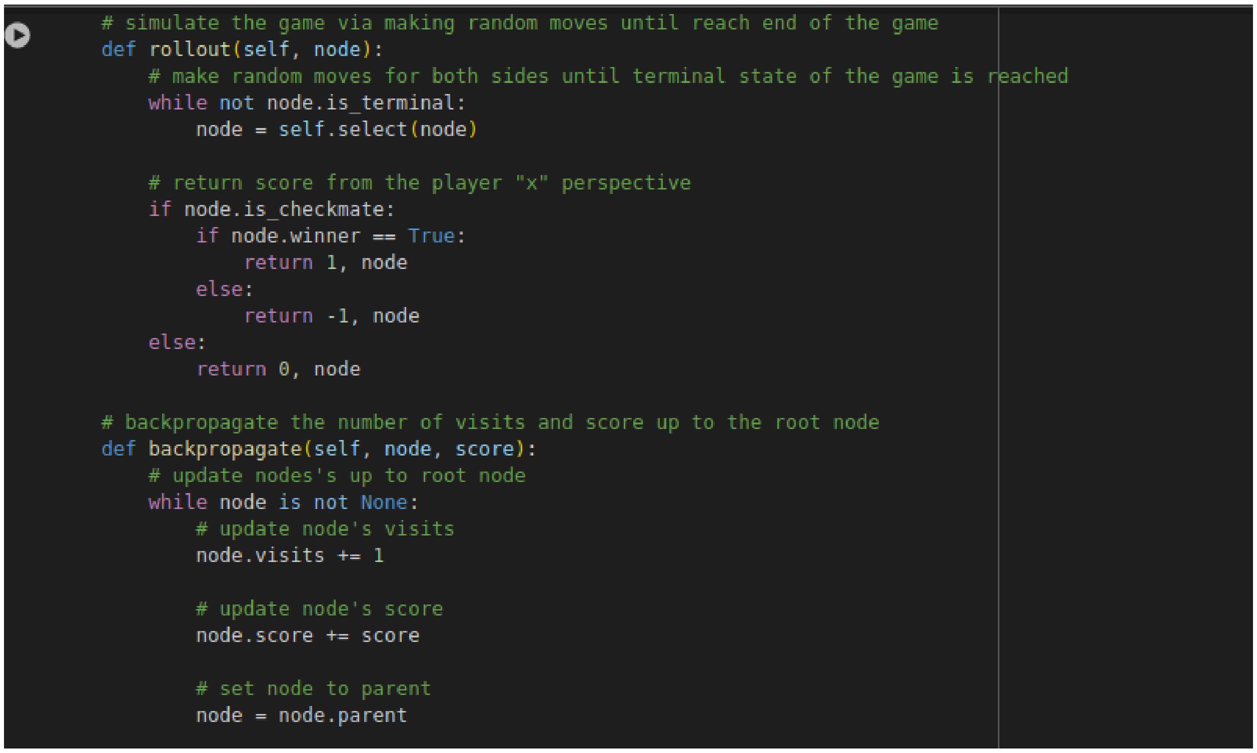
In this implementation, we took the five most promising moves from Stockfish instead of random moves and expanded the tree and played games using those moves only. The rest of the method remained the same. The screenshots are given below:



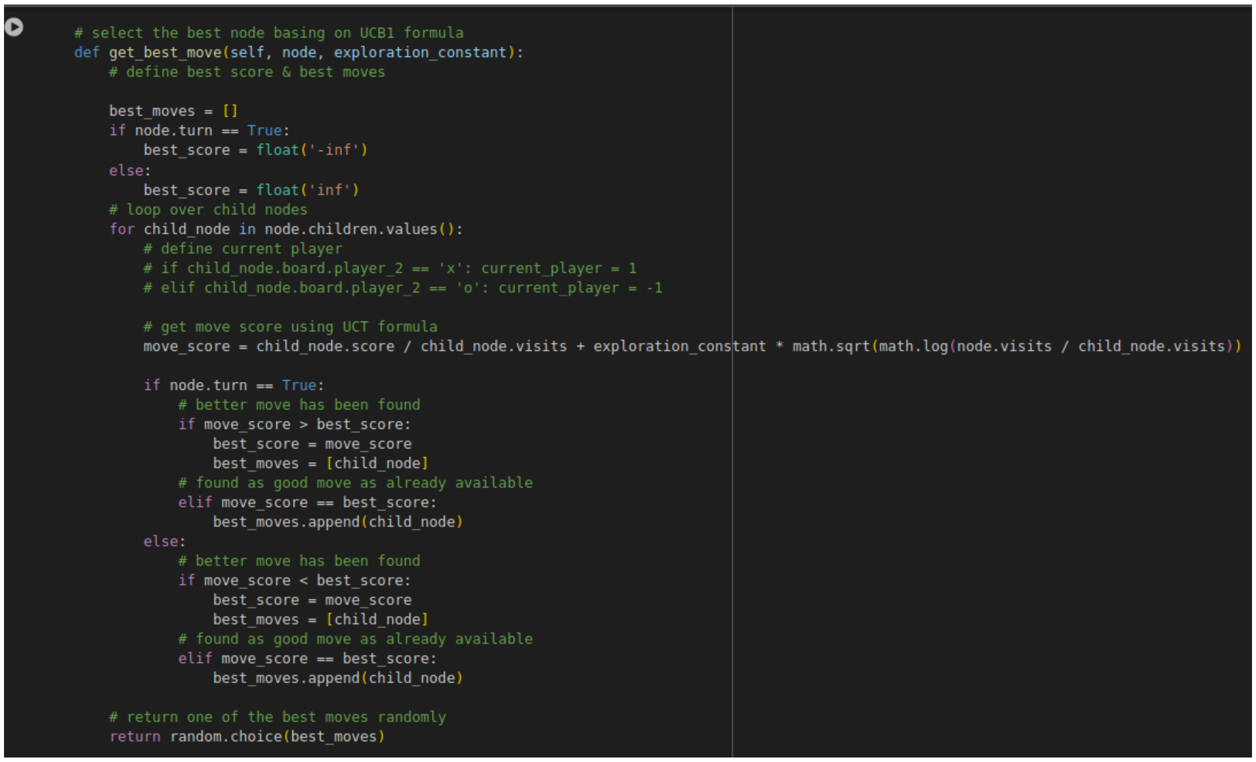
```
# MCTS class definition
class MCTS2():
    def __init__(self):
        self.stockfish = Stockfish(path="/home/sohaib/Downloads/stockfish-ubuntu-x86-64-vnni512")
        self.stockfish.set_depth(7)
    def search_existing_tree(self,node):
        for iteration in range(100):
            # select a node (selection phase)
            # node = self.select(self.root)

            # scrore current node (simulation phase)
            score, final_node = self.rollout(node)

            # backpropagate results
            self.backpropagate(final_node, score)

        return node

    # search for the best move in the current position
    def search(self,board):
        # create root node
        self.root = TreeNode2(board, None)
        # walk through 1000 iterations
        for iteration in range(200):
            # select a node (selection phase)
            # node = self.select(self.root)

            # scrore current node (simulation phase)
            score, final_node = self.rollout(self.root)

            # backpropagate results
            self.backpropagate(final_node, score)

        return self.root
```



```
    # select most promising node
    def select(self, node):
        # make sure that we're dealing with non-terminal nodes
        while not node.is_terminal:
            # case where the node is fully expanded
            if node.is_fully_expanded:
                node = self.get_best_move(node, 2)

            # case where the node is not fully expanded
            else:
                # otherwise expand the node
                return self.expand(node)

        # return node
        return node

    # expand node
    def expand(self, node):
        # generate legal states (moves) for the given node
        self.stockfish.set_fen_position(node.fen)
        top_moves = self.stockfish.get_top_moves(5)

        # loop over generated states (moves)
        for move in top_moves:
            move = move['Move']
            # make sure that current state (move) is not present in child nodes
            if move not in node.children:
                # create a new node
                new_board = chess.Board(node.fen)
                new_board.push_san(move)
                new_node = TreeNode2(new_board, node)
                del new_board

                # add child node to parent's node children list (dict)
                node.children[move] = new_node

                # case when node is fully expanded
                if len(top_moves) == len(node.children):
                    node.is_fully_expanded = True

                # return newly created node
                return new_node

        # debugging
        print('Should not get here!!!')
```



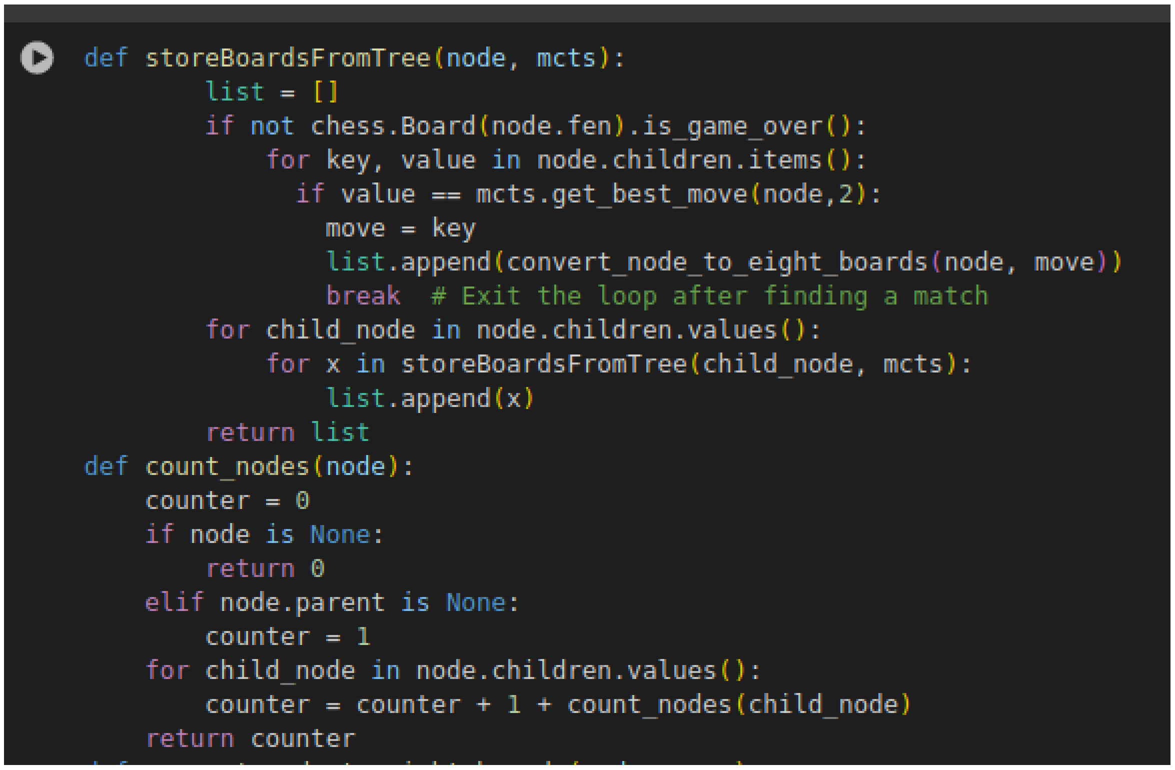
```python
    # simulate the game via making random moves until reach end of the game
    def rollout(self, node):
        # make random moves for both sides until terminal state of the game is reached
        while not node.is_terminal:
            node = self.select(node)

        # return score from the player "x" perspective
        if node.is_checkmate:
            if node.winner == True:
                return 1, node
            else:
                return -1, node
        else:
            return 0, node

    # backpropagate the number of visits and score up to the root node
    def backpropagate(self, node, score):
        # update nodes's up to root node
        while node is not None:
            # update node's visits
            node.visits += 1

            # update node's score
            node.score += score

            # set node to parent
            node = node.parent
```



```python
    # select the best node basing on UCB1 formula
    def get_best_move(self, node, exploration_constant):
        # define best score & best moves

        best_moves = []
        if node.turn == True:
            best_score = float('-inf')
        else:
            best_score = float('inf')
        # loop over child nodes
        for child_node in node.children.values():
            # define current player
            # if child_node.board.player_2 == 'x': current_player = 1
            # elif child_node.board.player_2 == 'o': current_player = -1

            # get move score using UCT formula
            move_score = child_node.score / child_node.visits + exploration_constant * math.sqrt(math.log(node.visits / child_node.visits))

            if node.turn == True:
                # better move has been found
                if move_score > best_score:
                    best_score = move_score
                    best_moves = [child_node]
                # found as good move as already available
                elif move_score == best_score:
                    best_moves.append(child_node)
            else:
                # better move has been found
                if move_score < best_score:
                    best_score = move_score
                    best_moves = [child_node]
                # found as good move as already available
                elif move_score == best_score:
                    best_moves.append(child_node)

        # return one of the best moves randomly
        return random.choice(best_moves)
```

When we ran this code, the results we saw were visibly better just within the tree. But now we had to train the policy network on this generated data. So we wrote code to convert all of the data in the tree to a form that could be used as inputs and outputs of a neural network.

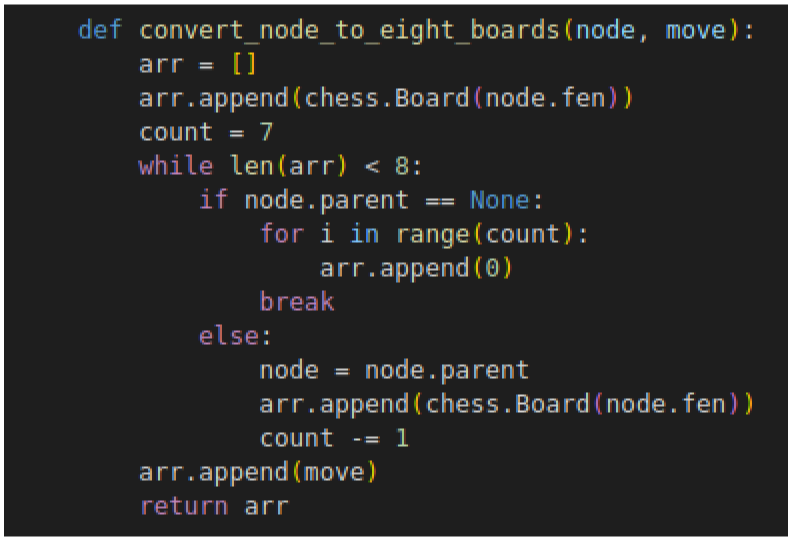
In order to do this we first got all the boards out from the nodes and put them into an array. Then we get the best moves from the tree and store them into an array with the corresponding indices.



```
def storeBoardsFromTree(node, mcts):
        list = []
        if not chess.Board(node.fen).is_game_over():
            for key, value in node.children.items():
              if value == mcts.get_best_move(node,2):
                move = key
                list.append(convert_node_to_eight_boards(node, move))
                break  # Exit the loop after finding a match
        for child_node in node.children.values():
            for x in storeBoardsFromTree(child_node, mcts):
                list.append(x)
        return list
def count_nodes(node):
    counter = 0
    if node is None:
        return 0
    elif node.parent is None:
        counter = 1
    for child_node in node.children.values():
        counter = counter + 1 + count_nodes(child_node)
    return counter
```

We also removed the boards which were the final states of the game as they would not have any best move.

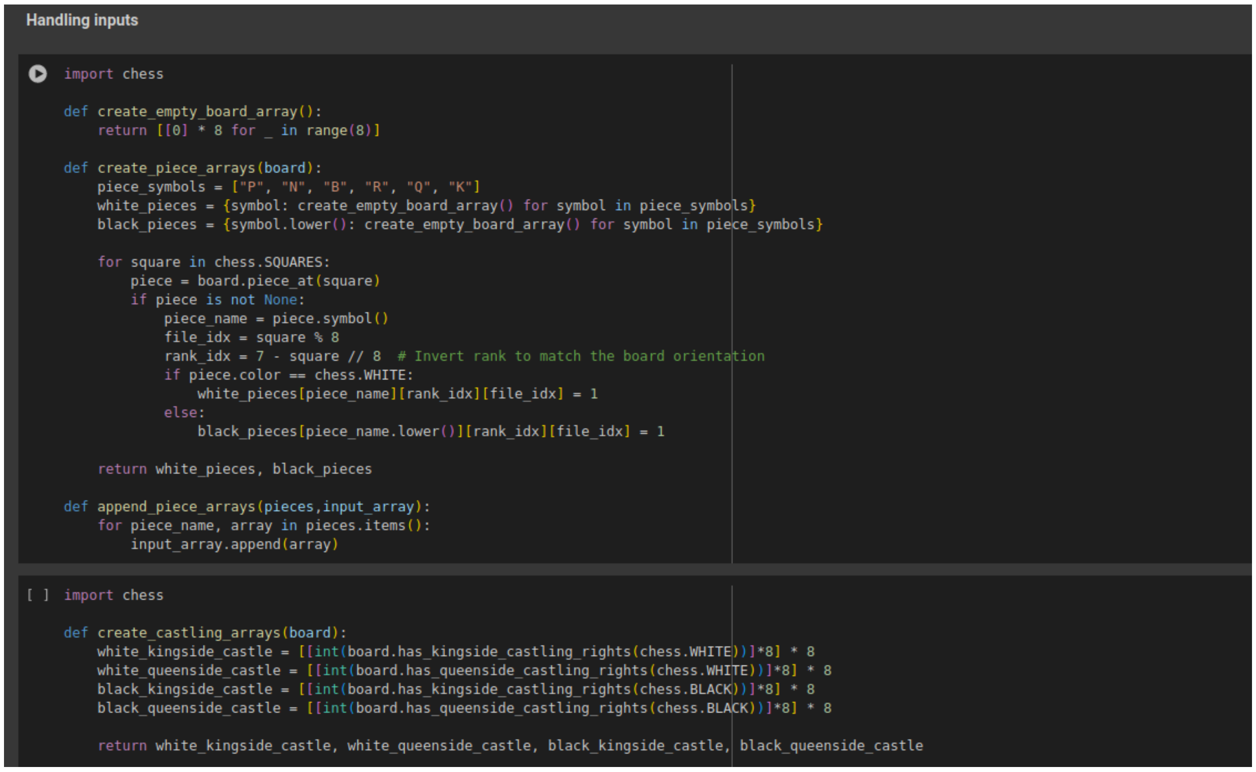
Now we have one board and one best move as the inputs and outputs. However, we need 8 boards including the previous 7 boards as well. This helps the model find patterns in moves and also helps identify threefold repetitions(draw),etc.
So we wrote another function that now stored 8 boards instead of one in the input array. The code is:



```python
def convert_node_to_eight_boards(node, move):
    arr = []
    arr.append(chess.Board(node.fen))
    count = 7
    while len(arr) < 8:
        if node.parent == None:
            for i in range(count):
                arr.append(0)
            break
        else:
            node = node.parent
            arr.append(chess.Board(node.fen))
            count -= 1
    arr.append(move)
    return arr
```

Now that we have the boards and the best moves corresponding to them, we need to convert them into a form easy to understand for the neural network.
We followed the exact same method as done in AlphaZero here and we encoded the boards in binary, just as is shown in the AlphaZero documentations above. For the output we followed AlphaZero as well and found all possible moves in every location and gave the network a total of 1969 possible outputs.
The code for this is given below:

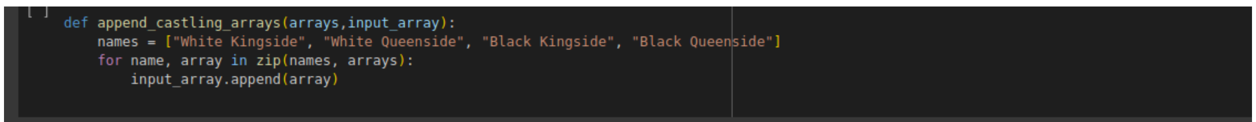
**Inputs**



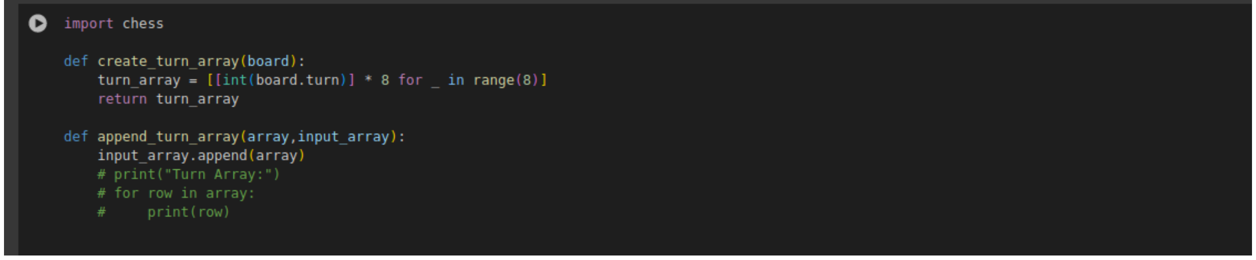
Handling inputs

```python
import chess

def create_empty_board_array():
    return [[0] * 8 for _ in range(8)]

def create_piece_arrays(board):
    piece_symbols = ["P", "N", "B", "R", "Q", "K"]
    white_pieces = {symbol: create_empty_board_array() for symbol in piece_symbols}
    black_pieces = {symbol.lower(): create_empty_board_array() for symbol in piece_symbols}

    for square in chess.SQUARES:
        piece = board.piece_at(square)
        if piece is not None:
            piece_name = piece.symbol()
            file_idx = square % 8
            rank_idx = 7 - square // 8  # Invert rank to match the board orientation
            if piece.color == chess.WHITE:
                white_pieces[piece_name][rank_idx][file_idx] = 1
            else:
                black_pieces[piece_name.lower()][rank_idx][file_idx] = 1

    return white_pieces, black_pieces

def append_piece_arrays(pieces,input_array):
    for piece_name, array in pieces.items():
        input_array.append(array)
```

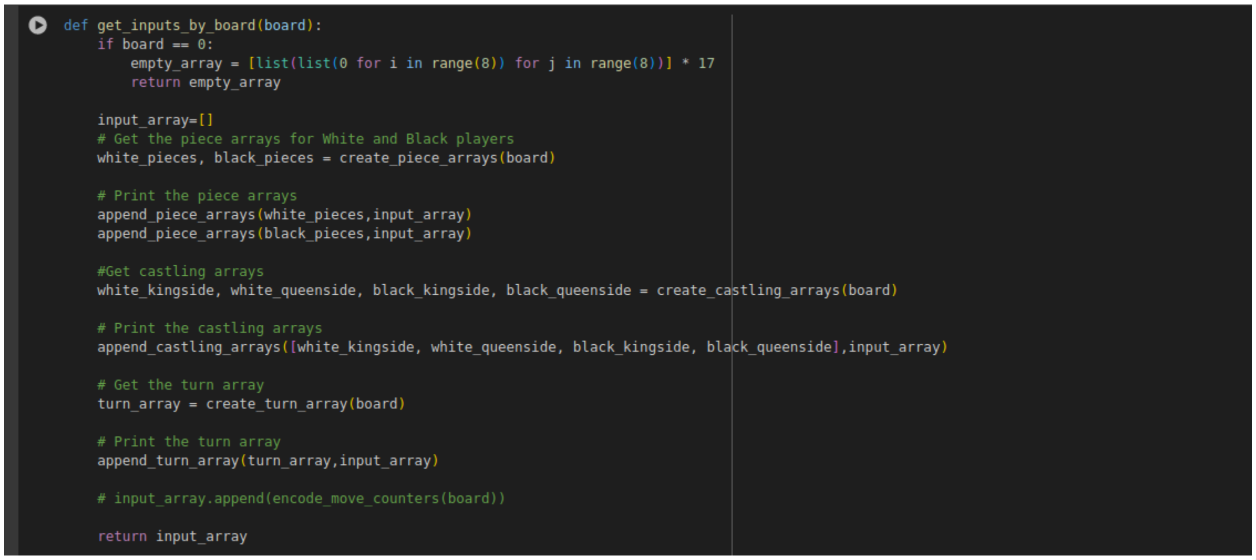
```python
[ ] import chess

def create_castling_arrays(board):
    white_kingside_castle = [[int(board.has_kingside_castling_rights(chess.WHITE))]*8] * 8
    white_queenside_castle = [[int(board.has_queenside_castling_rights(chess.WHITE))]*8] * 8
    black_kingside_castle = [[int(board.has_kingside_castling_rights(chess.BLACK))]*8] * 8
    black_queenside_castle = [[int(board.has_queenside_castling_rights(chess.BLACK))]*8] * 8

    return white_kingside_castle, white_queenside_castle, black_kingside_castle, black_queenside_castle
```



```python
def append_castling_arrays(arrays,input_array):
    names = ["White Kingside", "White Queenside", "Black Kingside", "Black Queenside"]
    for name, array in zip(names, arrays):
        input_array.append(array)
```

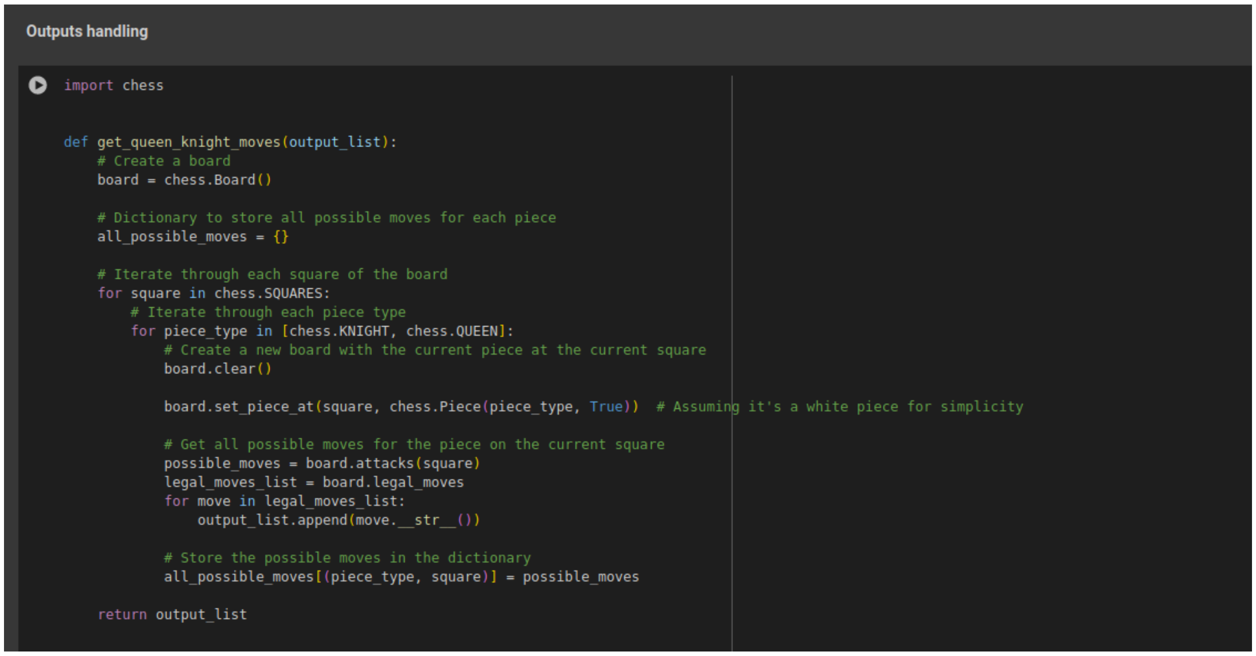


```python
import chess

def create_turn_array(board):
    turn_array = [[int(board.turn)] * 8 for _ in range(8)]
    return turn_array

def append_turn_array(array,input_array):
    input_array.append(array)
    # print("Turn Array:")
    # for row in array:
    #     print(row)
```



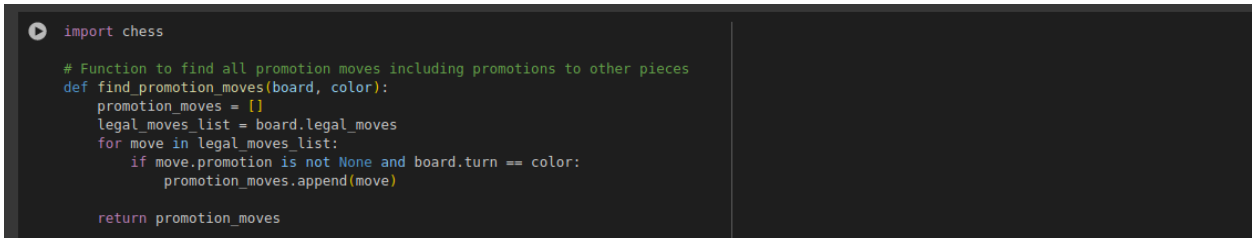
```python
def get_inputs_by_board(board):
    if board == 0:
        empty_array = [list(list(0 for i in range(8)) for j in range(8))] * 17
        return empty_array

    input_array=[]
    # Get the piece arrays for White and Black players
    white_pieces, black_pieces = create_piece_arrays(board)

    # Print the piece arrays
    append_piece_arrays(white_pieces,input_array)
    append_piece_arrays(black_pieces,input_array)

    #Get castling arrays
    white_kingside, white_queenside, black_kingside, black_queenside = create_castling_arrays(board)

    # Print the castling arrays
    append_castling_arrays([white_kingside, white_queenside, black_kingside, black_queenside],input_array)

    # Get the turn array
    turn_array = create_turn_array(board)

    # Print the turn array
    append_turn_array(turn_array,input_array)

    # input_array.append(encode_move_counters(board))

    return input_array
```

## Outputs

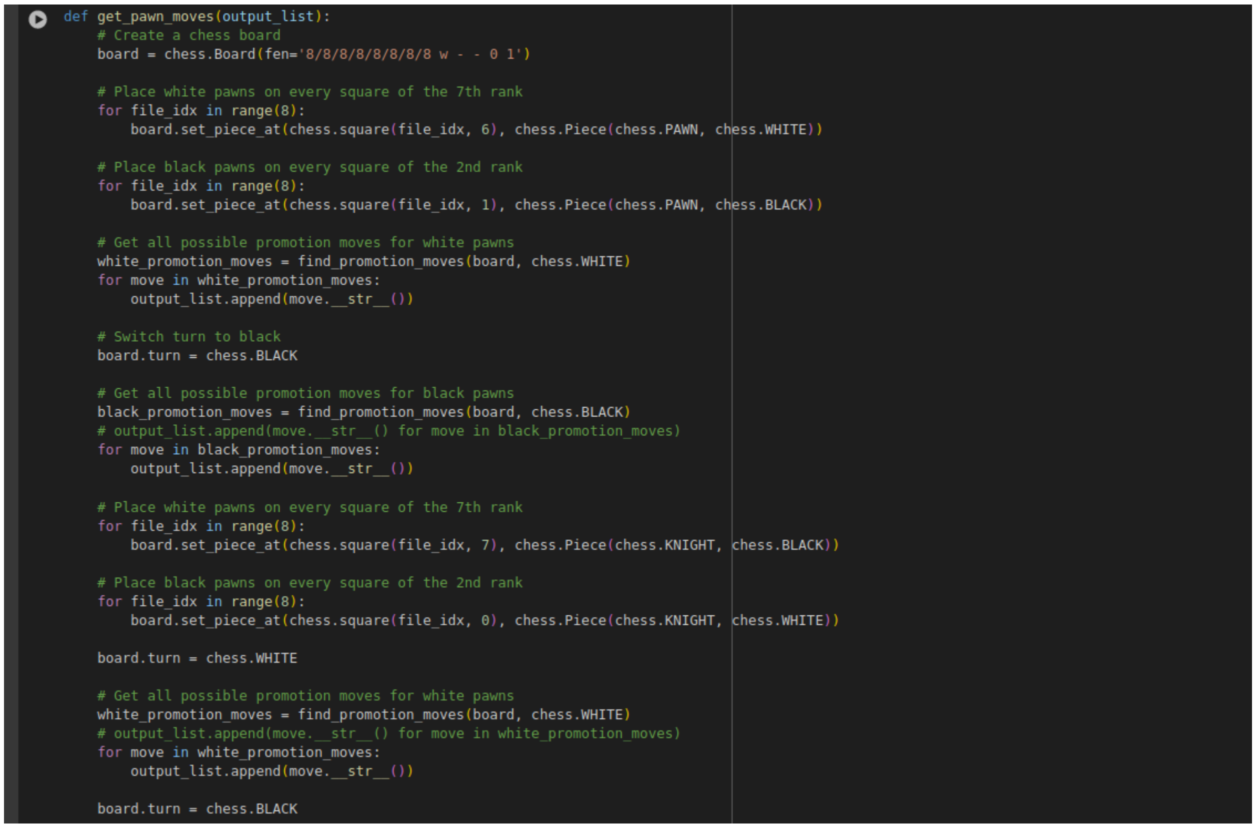


Outputs handling

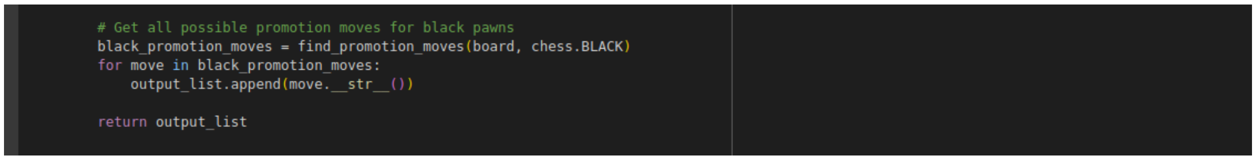
```python
import chess


def get_queen_knight_moves(output_list):
    # Create a board
    board = chess.Board()

    # Dictionary to store all possible moves for each piece
    all_possible_moves = {}

    # Iterate through each square of the board
    for square in chess.SQUARES:
        # Iterate through each piece type
        for piece_type in [chess.KNIGHT, chess.QUEEN]:
            # Create a new board with the current piece at the current square
            board.clear()

            board.set_piece_at(square, chess.Piece(piece_type, True))  # Assuming it's a white piece for simplicity

            # Get all possible moves for the piece on the current square
            possible_moves = board.attacks(square)
            legal_moves_list = board.legal_moves
            for move in legal_moves_list:
                output_list.append(move.__str__())

            # Store the possible moves in the dictionary
            all_possible_moves[(piece_type, square)] = possible_moves

    return output_list
```



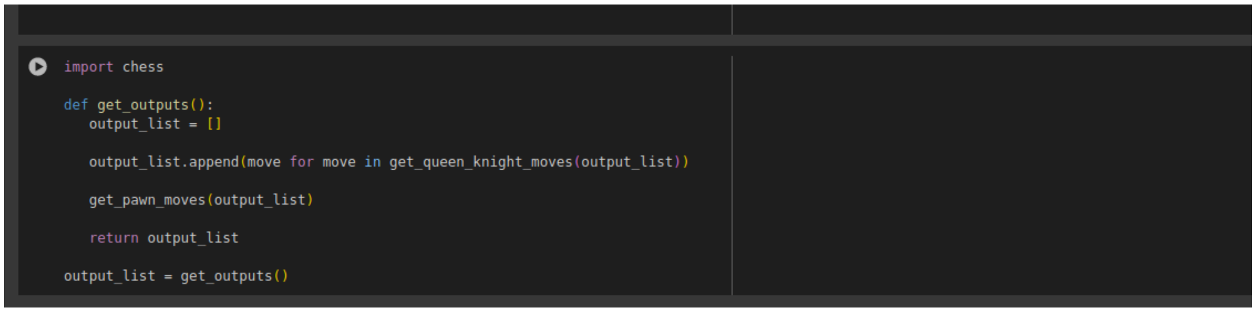
```python
import chess

# Function to find all promotion moves including promotions to other pieces
def find_promotion_moves(board, color):
    promotion_moves = []
    legal_moves_list = board.legal_moves
    for move in legal_moves_list:
        if move.promotion is not None and board.turn == color:
            promotion_moves.append(move)

    return promotion_moves
```



```python
def get_pawn_moves(output_list):
    # Create a chess board
    board = chess.Board(fen='8/8/8/8/8/8/8/8 w - - 0 1')

    # Place white pawns on every square of the 7th rank
    for file_idx in range(8):
        board.set_piece_at(chess.square(file_idx, 6), chess.Piece(chess.PAWN, chess.WHITE))

    # Place black pawns on every square of the 2nd rank
    for file_idx in range(8):
        board.set_piece_at(chess.square(file_idx, 1), chess.Piece(chess.PAWN, chess.BLACK))

    # Get all possible promotion moves for white pawns
    white_promotion_moves = find_promotion_moves(board, chess.WHITE)
    for move in white_promotion_moves:
        output_list.append(move.__str__())

    # Switch turn to black
    board.turn = chess.BLACK

    # Get all possible promotion moves for black pawns
    black_promotion_moves = find_promotion_moves(board, chess.BLACK)
    # output_list.append(move.__str__() for move in black_promotion_moves)
    for move in black_promotion_moves:
        output_list.append(move.__str__())

    # Place white pawns on every square of the 7th rank
    for file_idx in range(8):
        board.set_piece_at(chess.square(file_idx, 7), chess.Piece(chess.KNIGHT, chess.BLACK))

    # Place black pawns on every square of the 2nd rank
    for file_idx in range(8):
        board.set_piece_at(chess.square(file_idx, 0), chess.Piece(chess.KNIGHT, chess.WHITE))

    board.turn = chess.WHITE

    # Get all possible promotion moves for white pawns
    white_promotion_moves = find_promotion_moves(board, chess.WHITE)
    # output_list.append(move.__str__() for move in white_promotion_moves)
    for move in white_promotion_moves:
        output_list.append(move.__str__())

    board.turn = chess.BLACK
```

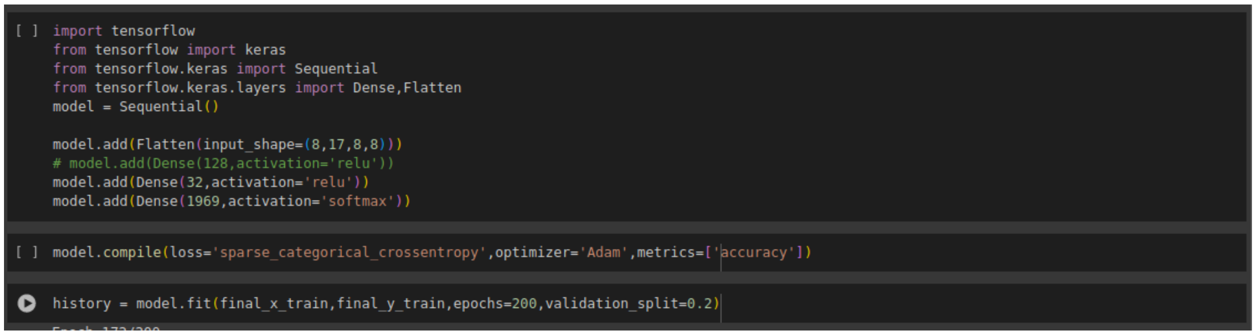


```python
    # Get all possible promotion moves for black pawns
    black_promotion_moves = find_promotion_moves(board, chess.BLACK)
    for move in black_promotion_moves:
        output_list.append(move.__str__())

    return output_list
```



```python
import chess

def get_outputs():
   output_list = []

   output_list.append(move for move in get_queen_knight_moves(output_list))

   get_pawn_moves(output_list)

   return output_list

output_list = get_outputs()
```

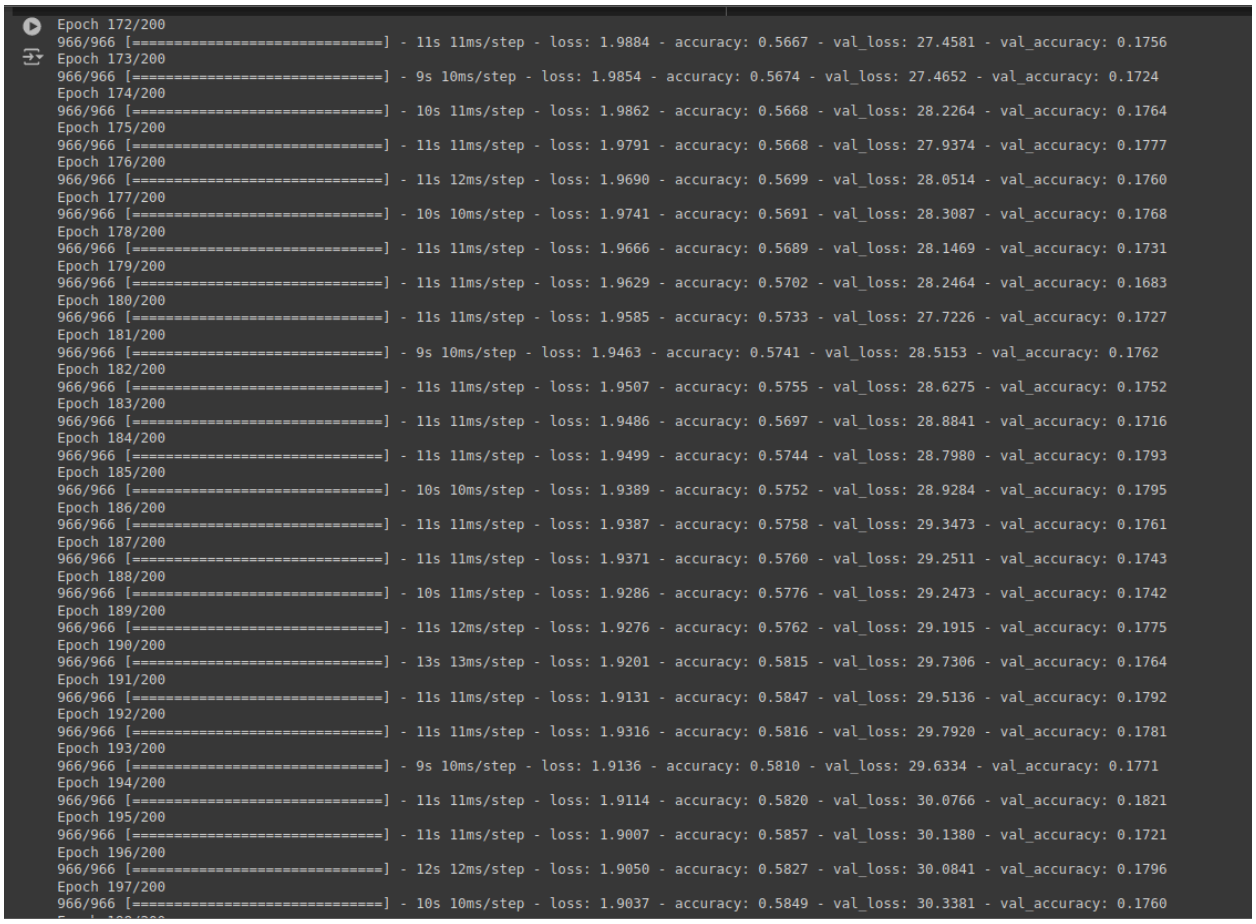
Now that we have the inputs and outputs we need, we simply trained a neural network on these inputs and outputs.



```
[ ] import tensorflow
    from tensorflow import keras
    from tensorflow.keras import Sequential
    from tensorflow.keras.layers import Dense,Flatten
    model = Sequential()

    model.add(Flatten(input_shape=(8,17,8,8)))
    # model.add(Dense(128,activation='relu'))
    model.add(Dense(32,activation='relu'))
    model.add(Dense(1969,activation='softmax'))
```

```
[ ] model.compile(loss='sparse_categorical_crossentropy',optimizer='Adam',metrics=['accuracy'])
```

```
history = model.fit(final_x_train,final_y_train,epochs=200,validation_split=0.2)
```



```
Epoch 172/200
966/966 [==============================] - 11s 11ms/step - loss: 1.9884 - accuracy: 0.5667 - val_loss: 27.4581 - val_accuracy: 0.1756
Epoch 173/200
966/966 [==============================] - 9s 10ms/step - loss: 1.9854 - accuracy: 0.5674 - val_loss: 27.4652 - val_accuracy: 0.1724
Epoch 174/200
966/966 [==============================] - 10s 11ms/step - loss: 1.9862 - accuracy: 0.5668 - val_loss: 28.2264 - val_accuracy: 0.1764
Epoch 175/200
966/966 [==============================] - 11s 11ms/step - loss: 1.9791 - accuracy: 0.5668 - val_loss: 27.9374 - val_accuracy: 0.1777
Epoch 176/200
966/966 [==============================] - 11s 12ms/step - loss: 1.9690 - accuracy: 0.5699 - val_loss: 28.0514 - val_accuracy: 0.1760
Epoch 177/200
966/966 [==============================] - 10s 10ms/step - loss: 1.9741 - accuracy: 0.5691 - val_loss: 28.3087 - val_accuracy: 0.1768
Epoch 178/200
966/966 [==============================] - 11s 11ms/step - loss: 1.9666 - accuracy: 0.5689 - val_loss: 28.1469 - val_accuracy: 0.1731
Epoch 179/200
966/966 [==============================] - 11s 11ms/step - loss: 1.9629 - accuracy: 0.5702 - val_loss: 28.2464 - val_accuracy: 0.1683
Epoch 180/200
966/966 [==============================] - 11s 11ms/step - loss: 1.9585 - accuracy: 0.5733 - val_loss: 27.7226 - val_accuracy: 0.1727
Epoch 181/200
966/966 [==============================] - 9s 10ms/step - loss: 1.9463 - accuracy: 0.5741 - val_loss: 28.5153 - val_accuracy: 0.1762
Epoch 182/200
966/966 [==============================] - 11s 11ms/step - loss: 1.9507 - accuracy: 0.5755 - val_loss: 28.6275 - val_accuracy: 0.1752
Epoch 183/200
966/966 [==============================] - 11s 11ms/step - loss: 1.9486 - accuracy: 0.5697 - val_loss: 28.8841 - val_accuracy: 0.1716
Epoch 184/200
966/966 [==============================] - 11s 11ms/step - loss: 1.9499 - accuracy: 0.5744 - val_loss: 28.7980 - val_accuracy: 0.1793
Epoch 185/200
966/966 [==============================] - 10s 10ms/step - loss: 1.9389 - accuracy: 0.5752 - val_loss: 28.9284 - val_accuracy: 0.1795
Epoch 186/200
966/966 [==============================] - 11s 11ms/step - loss: 1.9387 - accuracy: 0.5758 - val_loss: 29.3473 - val_accuracy: 0.1761
Epoch 187/200
966/966 [==============================] - 11s 11ms/step - loss: 1.9371 - accuracy: 0.5760 - val_loss: 29.2511 - val_accuracy: 0.1743
Epoch 188/200
966/966 [==============================] - 10s 11ms/step - loss: 1.9286 - accuracy: 0.5776 - val_loss: 29.2473 - val_accuracy: 0.1742
Epoch 189/200
966/966 [==============================] - 11s 12ms/step - loss: 1.9276 - accuracy: 0.5762 - val_loss: 29.1915 - val_accuracy: 0.1775
Epoch 190/200
966/966 [==============================] - 13s 13ms/step - loss: 1.9201 - accuracy: 0.5815 - val_loss: 29.7306 - val_accuracy: 0.1764
Epoch 191/200
966/966 [==============================] - 11s 11ms/step - loss: 1.9131 - accuracy: 0.5847 - val_loss: 29.5136 - val_accuracy: 0.1792
Epoch 192/200
966/966 [==============================] - 11s 11ms/step - loss: 1.9316 - accuracy: 0.5816 - val_loss: 29.7920 - val_accuracy: 0.1781
Epoch 193/200
966/966 [==============================] - 9s 10ms/step - loss: 1.9136 - accuracy: 0.5810 - val_loss: 29.6334 - val_accuracy: 0.1771
Epoch 194/200
966/966 [==============================] - 11s 11ms/step - loss: 1.9114 - accuracy: 0.5820 - val_loss: 30.0766 - val_accuracy: 0.1821
Epoch 195/200
966/966 [==============================] - 11s 11ms/step - loss: 1.9007 - accuracy: 0.5857 - val_loss: 30.1380 - val_accuracy: 0.1721
Epoch 196/200
966/966 [==============================] - 12s 12ms/step - loss: 1.9050 - accuracy: 0.5827 - val_loss: 30.0841 - val_accuracy: 0.1796
Epoch 197/200
966/966 [==============================] - 10s 10ms/step - loss: 1.9037 - accuracy: 0.5849 - val_loss: 30.3381 - val_accuracy: 0.1760
```

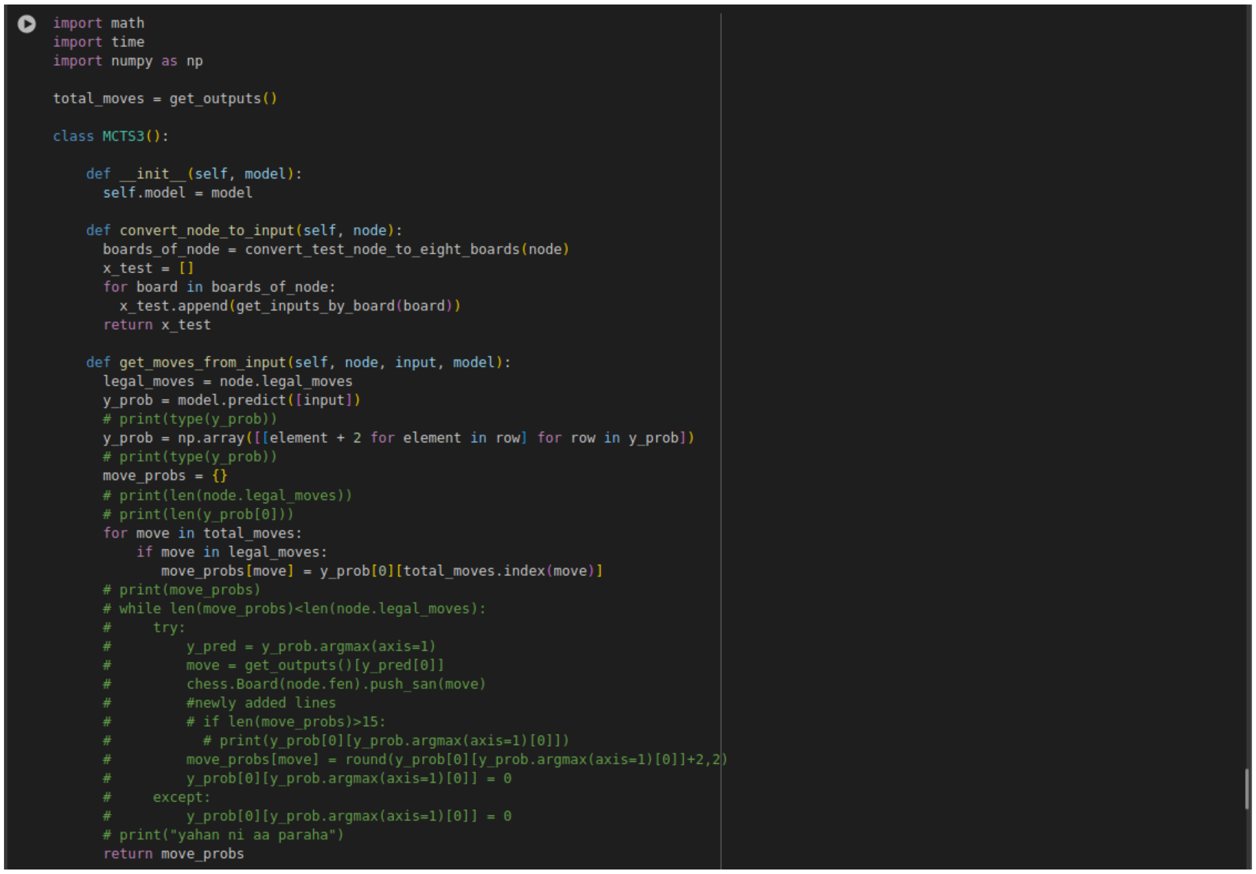
Now even though our implementation was correct, due to compute limitations we could get a very small monte carlo tree, and similarly, the model we trained was also trained for very little iterations.

However, even with these limitations, we analyzed the game played by this model and found that it was significantly better than the one that was played randomly, so the model had some understanding of the game.

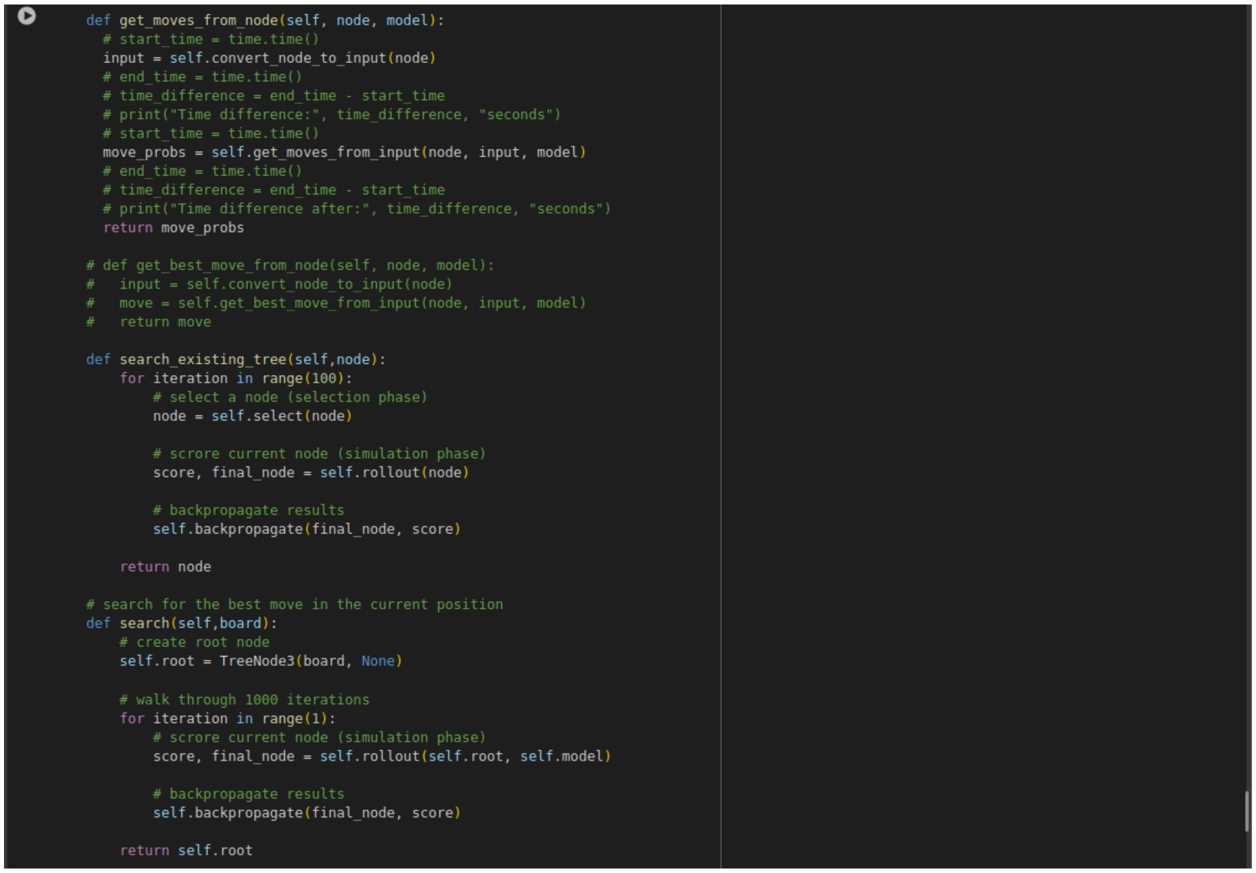
So we knew we were on the right path.

Now for the final step, we finally needed to use this model to create another monte carlo tree, so that we could now use this tree to start the iterative process of training and then creating the tree over and over again to improve the model.

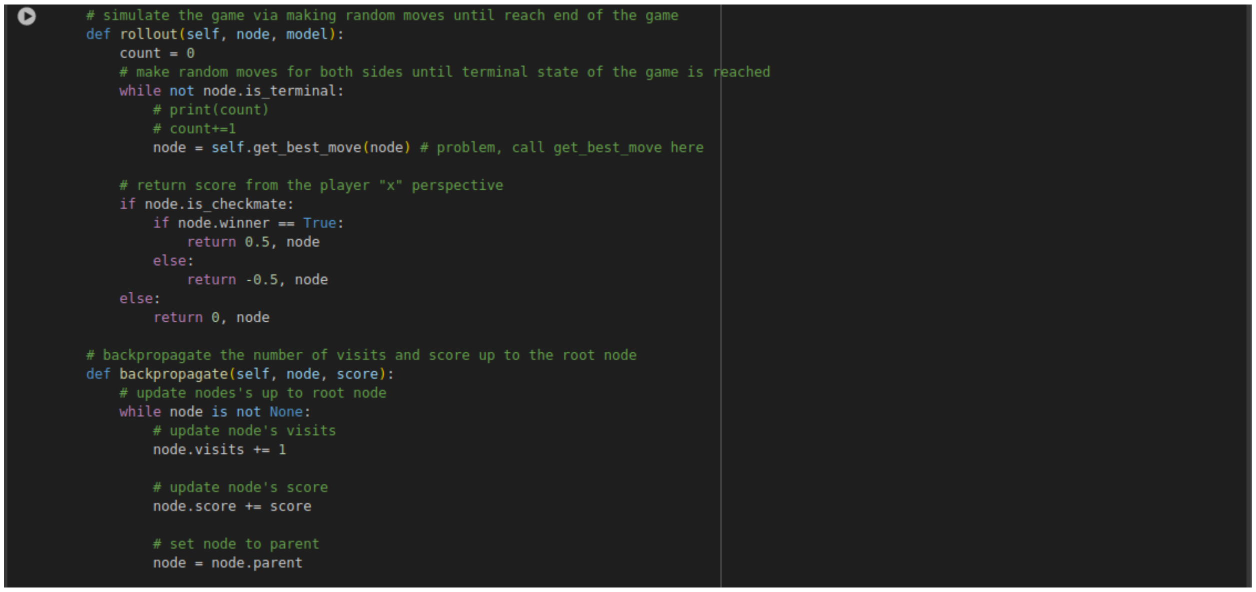
So to create the tree we wrote the code for this new tree once again using the model we had trained. The screenshot of the code is below:



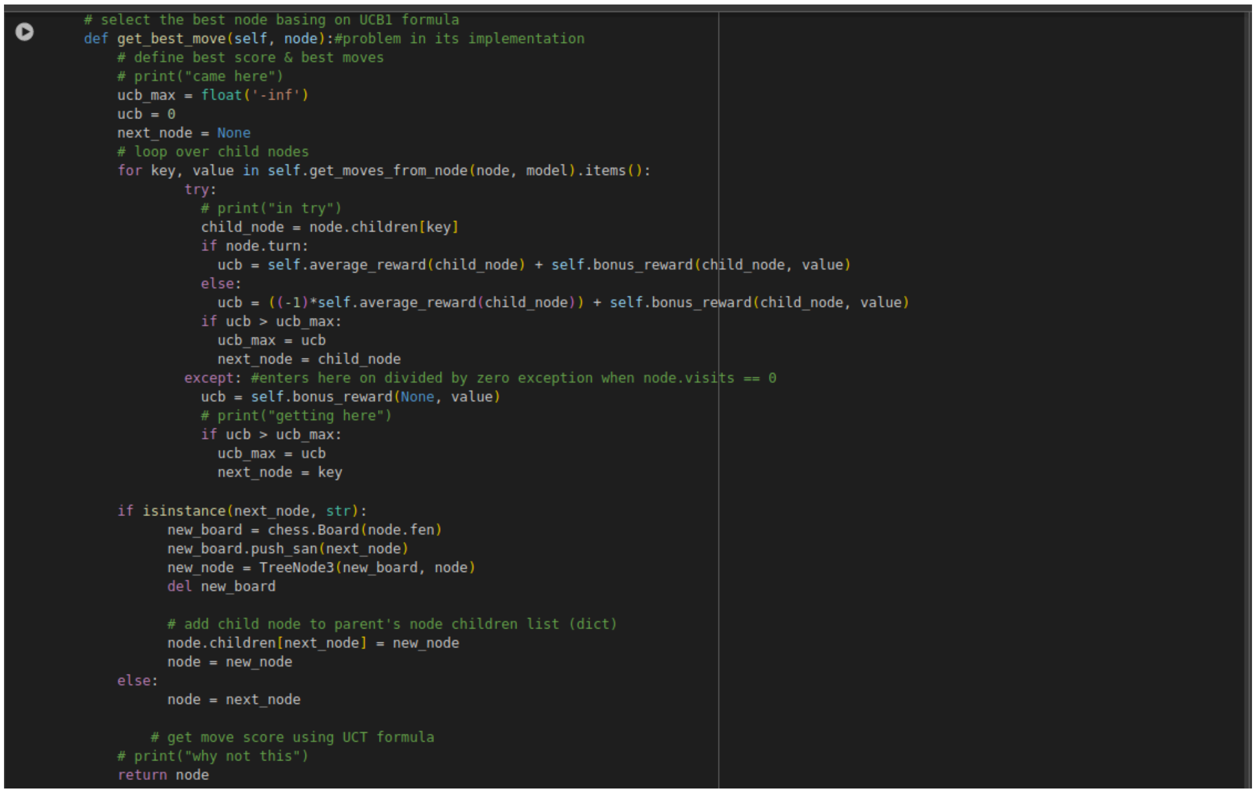
```
import math
import time
import numpy as np

total_moves = get_outputs()

class MCTS3():

    def __init__(self, model):
      self.model = model

    def convert_node_to_input(self, node):
      boards_of_node = convert_test_node_to_eight_boards(node)
      x_test = []
      for board in boards_of_node:
        x_test.append(get_inputs_by_board(board))
      return x_test

    def get_moves_from_input(self, node, input, model):
      legal_moves = node.legal_moves
      y_prob = model.predict([input])
      # print(type(y_prob))
      y_prob = np.array([[element + 2 for element in row] for row in y_prob])
      # print(type(y_prob))
      move_probs = {}
      # print(len(node.legal_moves))
      # print(len(y_prob[0]))
      for move in total_moves:
          if move in legal_moves:
             move_probs[move] = y_prob[0][total_moves.index(move)]
      # print(move_probs)
      # while len(move_probs)<len(node.legal_moves):
      #     try:
      #         y_pred = y_prob.argmax(axis=1)
      #         move = get_outputs()[y_pred[0]]
      #         chess.Board(node.fen).push_san(move)
      #         #newly added lines
      #         # if len(move_probs)>15:
      #           # print(y_prob[0][y_prob.argmax(axis=1)[0]])
      #         move_probs[move] = round(y_prob[0][y_prob.argmax(axis=1)[0]]+2,2)
      #         y_prob[0][y_prob.argmax(axis=1)[0]] = 0
      #     except:
      #         y_prob[0][y_prob.argmax(axis=1)[0]] = 0
      # print("yahan ni aa paraha")
      return move_probs
```



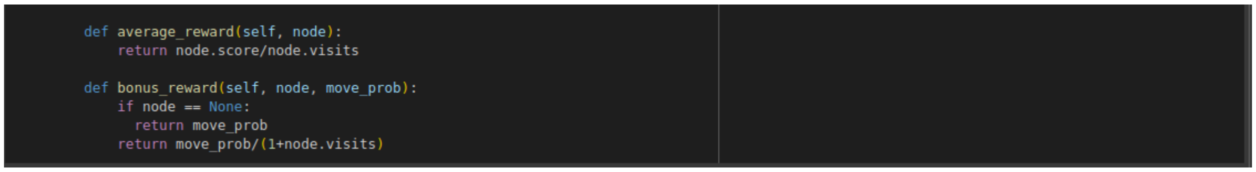
```python
def get_moves_from_node(self, node, model):
  # start_time = time.time()
  input = self.convert_node_to_input(node)
  # end_time = time.time()
  # time_difference = end_time - start_time
  # print("Time difference:", time_difference, "seconds")
  # start_time = time.time()
  move_probs = self.get_moves_from_input(node, input, model)
  # end_time = time.time()
  # time_difference = end_time - start_time
  # print("Time difference after:", time_difference, "seconds")
  return move_probs

# def get_best_move_from_node(self, node, model):
#   input = self.convert_node_to_input(node)
#   move = self.get_best_move_from_input(node, input, model)
#   return move

def search_existing_tree(self,node):
    for iteration in range(100):
        # select a node (selection phase)
        node = self.select(node)

        # scrore current node (simulation phase)
        score, final_node = self.rollout(node)

        # backpropagate results
        self.backpropagate(final_node, score)

    return node

# search for the best move in the current position
def search(self,board):
    # create root node
    self.root = TreeNode3(board, None)

    # walk through 1000 iterations
    for iteration in range(1):
        # scrore current node (simulation phase)
        score, final_node = self.rollout(self.root, self.model)

        # backpropagate results
        self.backpropagate(final_node, score)

    return self.root
```



```python
# simulate the game via making random moves until reach end of the game
def rollout(self, node, model):
    count = 0
    # make random moves for both sides until terminal state of the game is reached
    while not node.is_terminal:
        # print(count)
        # count+=1
        node = self.get_best_move(node) # problem, call get_best_move here

    # return score from the player "x" perspective
    if node.is_checkmate:
        if node.winner == True:
            return 0.5, node
        else:
            return -0.5, node
    else:
        return 0, node

# backpropagate the number of visits and score up to the root node
def backpropagate(self, node, score):
    # update nodes's up to root node
    while node is not None:
        # update node's visits
        node.visits += 1

        # update node's score
        node.score += score

        # set node to parent
        node = node.parent
```



```
# select the best node basing on UCB1 formula
def get_best_move(self, node):#problem in its implementation
    # define best score & best moves
    # print("came here")
    ucb_max = float('-inf')
    ucb = 0
    next_node = None
    # loop over child nodes
    for key, value in self.get_moves_from_node(node, model).items():
            try:
              # print("in try")
              child_node = node.children[key]
              if node.turn:
                ucb = self.average_reward(child_node) + self.bonus_reward(child_node, value)
              else:
                ucb = ((-1)*self.average_reward(child_node)) + self.bonus_reward(child_node, value)
              if ucb > ucb_max:
                ucb_max = ucb
                next_node = child_node
            except: #enters here on divided by zero exception when node.visits == 0
              ucb = self.bonus_reward(None, value)
              # print("getting here")
              if ucb > ucb_max:
                ucb_max = ucb
                next_node = key

    if isinstance(next_node, str):
          new_board = chess.Board(node.fen)
          new_board.push_san(next_node)
          new_node = TreeNode3(new_board, node)
          del new_board

          # add child node to parent's node children list (dict)
          node.children[next_node] = new_node
          node = new_node
    else:
          node = next_node

        # get move score using UCT formula
    # print("why not this")
    return node
```

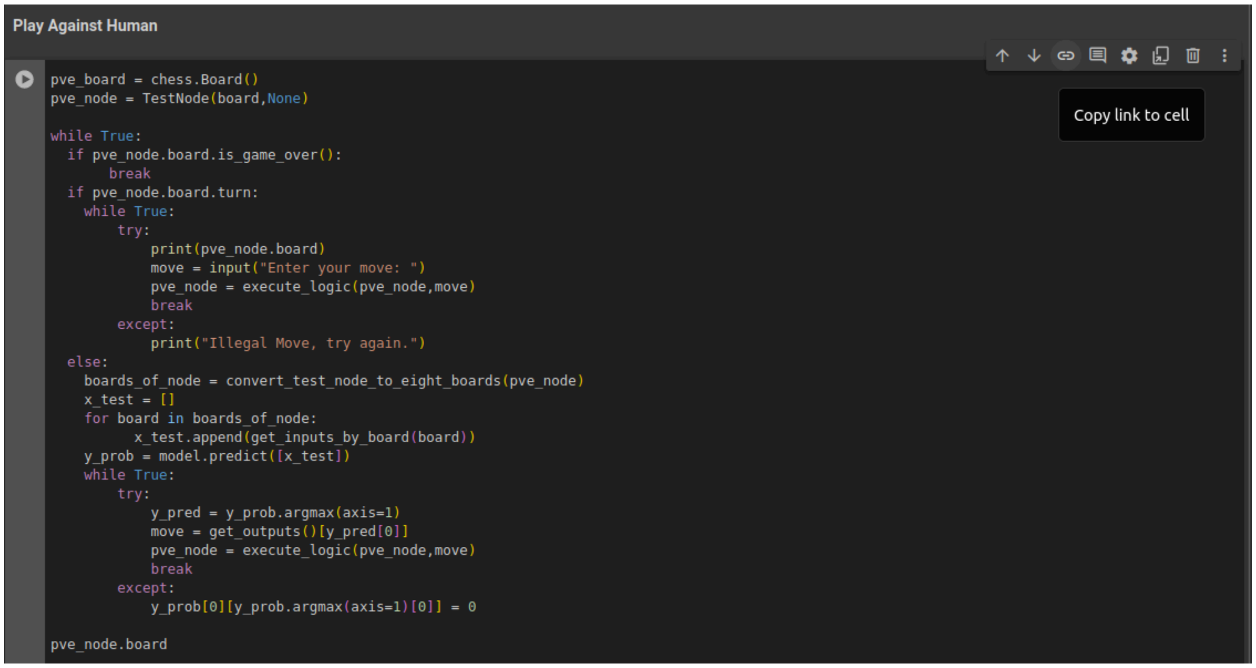


```
def average_reward(self, node):
    return node.score/node.visits

def bonus_reward(self, node, move_prob):
    if node == None:
      return move_prob
    return move_prob/(1+node.visits)
```

This code creates a new monte carlo tree from scratch using the model we trained as the policy network. But since we had already used a lot of our compute resources and this was a very compute heavy process, we could not run this code fully and were not able to achieve the end goal but the output of the current work were very visible and we are on the right path for sure.

## Final Results

Now we have the trained network, a new monte carlo tree from that network, and code to convert that tree to a new model. We also wrote code to make the model play a game against a human which can be modified to play against another model.



```
pve_board = chess.Board()
pve_node = TestNode(board,None)

while True:
  if pve_node.board.is_game_over():
      break
  if pve_node.board.turn:
    while True:
        try:
            print(pve_node.board)
            move = input("Enter your move: ")
            pve_node = execute_logic(pve_node,move)
            break
        except:
            print("Illegal Move, try again.")
  else:
    boards_of_node = convert_test_node_to_eight_boards(pve_node)
    x_test = []
    for board in boards_of_node:
          x_test.append(get_inputs_by_board(board))
    y_prob = model.predict([x_test])
    while True:
        try:
            y_pred = y_prob.argmax(axis=1)
            move = get_outputs()[y_pred[0]]
            pve_node = execute_logic(pve_node,move)
            break
        except:
            y_prob[0][y_prob.argmax(axis=1)[0]] = 0

pve_node.board
```

Therefore, now we have the full code for the bot and we can easily implement it at a higher level if we have the compute resources.

**References:**

1. Giraffe: Using Deep Reinforcement Learning to Play Chess
2. Using Reinforcement Learning in Chess Engines
3. Reinforcement Learning in an AdaptableChess Environment for DetectingHuman-understandable Concepts
4. Investigations into Playing Chess Endgames using Reinforcement Learning.
5. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm
6. IMPLEMENTATION OF A CHESS GAME PLAYING SYSTEM USING MACHINE LEARNING
7. Application of Q-Learning and RBF Network in Chinese Chess Game System
8. AlphaGo Zero & AlphaZero Mastering Go, Chess and Shogi wihtout human knowledge
9. A general reinforcement learningalgorithm that masters chess, shogi,and Go through self-play
10. AlphaZero Vs StockFish – A Review of Chess Engines
11. Neural Networks for Chess